BEGINNING

Z 80

MACHINE CODE

Syd Wyncoop

A STUDY IN NUMBERS

Number Systems (Bases)

Why would we want to study numbers? Because, our computers only understand numbers. You may saying, "My computer knows Basic" or any of several other languages but, the truth of the matter is that it does not know or understand any language other than the assembly language of the Central Processing Unit (CPU).

What is this fool (me) talking about? All languages used by any computer, except assembly, are referred to as 'high level' languages. High level languages come in two varieties, compiled and interpreted. Compiled languages are converted (compiled) into the CPU's assembly language and run as machine code and therefore quicker than interpreted languages.

Interpreted languages are interpreted into assembly language at the time of execution. They still run as machine code but, much slower due to the interpreter processing time. Interpreted languages also run from a library of routines, which are slower by their very nature.

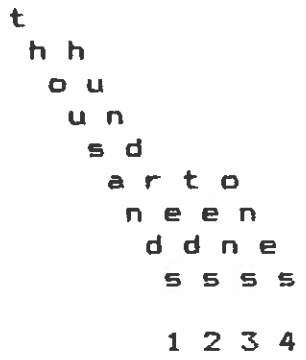
The important point of this is that all programs are stored in the computer memory as numbers. Look at your Sinclair character tables. The Basic command PRINT is really the number 245. Since all programs are numbers, it behooves us to know a little bit about them. I don't know about you but, I hate machines that are smarter than I am!

We do not need to learn assembly programming to use our computers however, understanding the numbers and why different bases are used will help us be more efficient programmers. Of course many of you may have desired to know more about machine code and been afraid to tackle it. After all, those long Hex (whatever that is) dumps in the magazines seldom make sense. Or maybe you have seen those Hexidecimal numbers and wondered why anyone would resort to such mutterings! If this discribes you, then perhaps the following study can be of help.

If we wish to get intimate with some of the inner workings of our computer or before attempting to learn any Assembly Language (machine code), you will need to thoroughly understand three number systems or bases as they are properly called. There is a fourth base, Octal, which I will not discuss here as you will seldom encounter it. Octal will be found in the programming guides published by chip manufactures and can be useful however, we can do quite well without it for this study. Each base will be discussed separately however, I will make references back to Decimal, as that is the one we all use dailey.

Decimal (Base 10)

The first is Decimal (dec), or base 10 numbers. We are all familiar with decimal as we use it every day to count our bags of money and for various other tasks. Most of us learned in school that the columns of digits represent ones, tens, hundreds, etc. For example:



What you may not recall is, each column represents 10 (the base)raised to the power of the number of the column as counted from right to left, starting at zero. Hows that for doubletalk?

Follow through this example, to see that 1234 decimal really means:

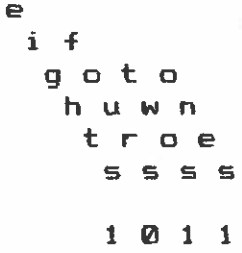
```
10^0 =    1*4 =    4
10^1 =   10*3 =   30
10^2 =  100*2 =  200
10^3 = 1000*1 = 1000
                -----
Total=           1234
```

Study this carefully as it is the easiest example we will have and it must be understood or the rest will really seem like Greek to you.

Before continuing, it bears mentioning that all bases are represented by the digits 0 to Base-1. Base 10 is therefore represented by the digits 0 to 9. It is these digits which then represent a mutiple of a power of the base, as above (10^0*4). No digit can be greater than Base-1 because at the point it equals the Base there is a carry to the next column. For instance, in base 10, 9+1=0 and carry 1 to next column. All of this should be familiar to you but if you are like me, you haven't given it much thought since school. No,I wont say how long ago that was!

Binary (Base 2)

Now for the hard stuff, Binary (bin), or base 2. Following our discussion of the last paragraph, we can only represent binary numbers with the digits 0 and 1 (remember base-1). This means our columns also must have meanings other than ones, tens, etc. They now become ones, twos, fours, eights, sixteens, etc., which are the powers of 2 (our base) instead of powers of 10. For example:



As in our previous example, 1011 really means:

```
2^0 = 1*1 = 1
2^1 = 2*1 = 2
2^2 = 4*0 = 0
2^3 = 8*1 = 8
            ---
Total=      11 decimal
```

Now you know why we count in decimal! You thought it was because we have ten fingers. Imagine having hands with two fingers on each hand. Binary would then seem as easy as decimal. If you have any difficulty with this, go back to the discussion on decimal and compare it with this one. Only the base is different.

What you have just learned is how to convert binary to decimal. The procedure for converting decimal to binary is similar. Briefly, divide your number by the largest power of two not larger than your number. You continue this process with successively smaller powers of two until you have reached 2^0, at which time there should not be a remainder. Write down (left to right) a 1 when the division is possible and a 0 when not possible. Using our example of 11:

Step	Do	Result
1.	2^4=16 and 2^3=8 therefore, 2^3 or 8 is the number we want.	
2.	2^3=8 and 11/8=1 remainder 3	1
3.	2^2=4 and 3/4=0 remainder 3	10
4.	2^1=2 and 3/2=1 remainder 1	101
5.	2^0=1 and 1/1=1 remainder 0	1011
6.	We have now converted 11 decimal to 1011 binary	

Some of you may be wondering what the point of all this is. After all I barely passed math in school, why bother with this now? The point is, while decimal is more comfortable for us humans, binary better represents how our computer 'thinks'.

An explanation of the CPU is in order. This is background only to give you some understanding why the 'smart' computer doesn't understand decimal. The CPU (which is the Z80 in our Sinclairs) is merely a collection of transistors and transistors are simply electronic switches. For those of you who know better, please bear with me, my end will justify the extreme over simplicatiom. We all know that a switch can either be on or off. Binary allows us to represent the on/off states with one and zero, respectively. Not quite perfect but it works!

Most binary numbers you will see, will have eight digits or some multiple thereof. This is accomplished by padding out the number with leading zeros. For instance, 1011 binary would normally be written as 00001011b. The reason for eight Binary digITS (bits) is that eight bits make one byte.

A byte is not what your neighbors' dog puts on you. A byte is the 'wordsize' of your CPU. A word is the number of bits handled as a complete unit by the CPU and is commonly referred to as a byte. The Z80 is an eight bit CPU, therefore one byte=eight bits. Words and bytes are not exactly the same, but will suffice here. You do not need to understand the internal workings of the CPU in order to understand binary numbers.

Binary is most useful for graphics in Basic or masks in assembly. Unless you decide to learn assembly, you will seldom

work directly in Binary due to the difficulty keeping the digits correct, which leads us to the next base....

Hexadecimal (Base 16)

Now that we have mastered binary we can dig into the last of the three number systems. Hexidecimal (hex) or base 16 is used because it works very well as a shorthand for that awful binary. I have provided several charts for easy conversion from hex to decimal and hex to binary or vice versa. So don't leave now, things are about to make sense.

One challenge we have with hex is that there are not enough digits for 0 to base-1. We have only 0 to 9 which will work fine for the first ten digits. We need to improvise for the last six digits and someone far wiser than me (us?) has already solved our dilemna. The digits needed to represent 10 to 15 are A to F. The sixteen hex digits are now 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E & F.

Now that we have the digits, we need to recognize the form taken by hex numbers. Just as binary are usually eight digits long, hex numbers are always two (or some multiple of two) digits long. This makes counting in hex as follows:

```
00,01,02,03,...,0A,0B,0C,0D,0E,0F,
10,11,12,13,.............,1D,1E,1F,
20,21,.........................,2F,
30,.............................,3F,
.................................,
F0,.............................,FF
```

A close look at the hex to decimal conversion chart will make this much clearer.

You may occasionally see hex numbers that have an odd number of digits. The first digit will be a zero and the second a letter, such as 0FFh. I do not use a leading zero however, be aware some assemblers require it, therefore you may see it.

There is no need to go into the math needed to convert between hex and decimal as the chart provided will serve the purpose much better, easier and faster. Those of you interested in working out the details need only follow the examples for decimal and binary. Keep in mind the base is now 16 therefore,the columns of digits will represent ones, sixteens, two hundredfifty-sixes, etc.

All this brings us to what I mean by hex being a shorthand for binary and the reason we are even interested in hex. A close look at the hex to binary conversion chart will make this more obvious. You can readily see that four bits can be represented by one hex digit. For those of you who understood my ramblings about bytes, four bits is a small byte (also known in some circles as a nybble). Could I make this up? Therefore, we can represent any eight bit byte with only two hex digits.

You may recognize that this is not that much better than decimal however, decimal cannot be converted to binary with the same ease as hex. Also, numbers larger than 255 will really create some headaches that hex helps solve (more on this in a minute).

We now know how to write numbers in three bases, decimal, binary, and hex. In order to avoid confusion we need to make a proper designation of each. Thoughout this study we will suffix all binary numbers with a 'b' (10000100b) and all hex numbers with a 'h' (25h). Occasionally, you may see a 'd' suffix on decimal numbers however, it is not needed as decimal is the default. We will do this even though some numnbers can obviously only be hex (FFh). You must always be careful to follow this notation or you will create unneccessary confusion for all.

You should take some time to practice using all the charts. Also, practice simple arithmetic in each base (add, sub, mult & div). You can use the charts to check your answers. Before long you will be thinking in hex and binary as easily as you now do in decimal.

Ok, let's look at how numbers larger than 255 are stored and handled by the CPU.

That's twice I mentioned 255 without an explanation. The reason 255 is a magic number is because it is base-1 for base 256 numbers. I am not going to boggle your mind with this number system as it is not needed by us, only the CPU uses it. Why in the world does the silly CPU use base 256? Let's go back to our discusion of binary, bytes and related whatevers. Remember, we padded out our binary numbers to eight digits as one byte for the Z80 is eight bits? The reason the CPU uses base 256 is 2^8 = 256. Voila! On your own, figure out what 11111111b means in decimal.

We can actually store numbers between 0 and 65535 by tieing two bytes together. This is done by the CPU automatically to generate addresses. The second byte is increased by a factor of 256 as this is the number that generates a carry out of the first byte. For example:

```
  1 1 1 1 1 1 1 1
+               1
----------------
1 0 0 0 0 0 0 0 0
```

Remember, 1+1 binary = 0 and carry 1 just as 9+1 does in decimal. Work out the above problem yourself to see how the result is achieved. You will notice we now have a number which is nine digits long. In truth, the actual number is 000000010000000b because we are tieing two bytes together and each byte is padded out to eight digits. Since our number is now sixteen bits long, the largest number we can store is 2^16-1 or 65535.

You can now see why we need a shorthand for binary. Today, we rarely work with large binary numbers as it is too easy to err. We will not discuss binary numbers larger than eight digits either. If you decide to learn an assembly language, you will probably only use binary when working with the logic instructions. You may find it interesting that early programmers had to use only binary numbers and they were entered from a panel of switches instead of a keyboard. Today, using machine code is duck soup and hex is much easier to work with.

What did I mean, 'tie two bytes together and increase one of them by a factor of 256?' Let's assume our number is stored in byte 1 and byte 2. The formula to recover our number is:

```
Peek byte 1 + Peek byte 2 * 256
```

Looks a little familiar? You probably have seen something similiar before and did not know what was happening. The byte we increase by the factor of 256 is called the high byte which makes the other byte the low byte. Using high/low nomenclature, our formula becomes:

Peek low byte + Peek high byte * 256

One pecularity designed into the CPU is that contrary to the number systems we have discussed, the CPU stores the low byte first. This must be kept in mind or you will not at all get the results you were trying to achieve. Scan the list of system variables in your Sinclair manual and use this formula on some of the 2 byte variables. The results are the address at which that area of memory begins.

Review some of the areas of your Sinclair manual that did not make sense before. Especially the chapters on number systems, machine code, system variables, memory, and the appendices. There is a wealth of information there however, it is presented so poorly that it may not have made sense before. Then compare notes with this study and you will be well on the way to understanding machine code.

--END--

Syd Wyncoop
2107 SE 155th
Portland, Or 97233
503)760-7786

Hex-Bin Conversions

Hex	Bin
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Chart 7

Hex/Dec Conversions

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	Ø	1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14	15
1	16	17	18	19	2Ø	21	22	23	24	25	26	27	28	29	3Ø	31
2	32	33	34	35	36	37	38	39	4Ø	41	42	43	44	45	46	47
3	48	49	5Ø	51	52	53	54	55	56	57	58	59	6Ø	61	62	63
4	64	65	66	67	68	69	7Ø	71	72	73	74	75	76	77	78	79
5	8Ø	81	82	83	84	85	86	87	88	89	9Ø	91	92	93	94	95
6	96	97	98	99	1ØØ	1Ø1	1Ø2	1Ø3	1Ø4	1Ø5	1Ø6	1Ø7	1Ø8	1Ø9	11Ø	111
7	112	113	114	115	116	117	118	119	12Ø	121	122	123	124	125	126	127
8	128	129	13Ø	131	132	133	134	135	136	137	138	139	14Ø	141	142	143
9	144	145	146	147	148	149	15Ø	151	152	153	154	155	156	157	158	159
A	16Ø	161	162	163	164	165	166	167	168	169	17Ø	171	172	173	174	175
B	176	177	178	179	18Ø	181	182	183	184	185	186	187	188	189	19Ø	191
C	192	193	194	195	196	197	198	199	2ØØ	2Ø1	2Ø2	2Ø3	2Ø4	2Ø5	2Ø6	2Ø7
D	2Ø8	2Ø9	21Ø	211	212	213	214	215	216	217	218	219	22Ø	221	222	223
E	224	225	226	227	228	229	23Ø	231	232	233	234	235	236	237	238	239
F	24Ø	241	242	243	244	245	246	247	248	249	25Ø	251	252	253	254	255

Chart 7

Signed Numbers--Hex/Dec Conversions

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	Ø	1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14	15
1	16	17	18	19	2Ø	21	22	23	24	25	26	27	28	29	3Ø	31
2	32	33	34	35	36	37	38	39	4Ø	41	42	43	44	45	46	47
3	48	49	5Ø	51	52	53	54	55	56	57	58	59	6Ø	61	62	63
4	64	65	66	67	68	69	7Ø	71	72	73	74	75	76	77	78	79
5	8Ø	81	82	83	84	85	86	87	88	89	9Ø	91	92	93	94	95
6	96	97	98	99	1ØØ	1Ø1	1Ø2	1Ø3	1Ø4	1Ø5	1Ø6	1Ø7	1Ø8	1Ø9	11Ø	111
7	112	113	114	115	116	117	118	119	12Ø	121	122	123	124	125	126	127
8	-128	-127	-126	-125	-124	-123	-122	-121	-12Ø	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-11Ø	-1Ø9	-1Ø8	-1Ø7	-1Ø6	-1Ø5	-1Ø4	-1Ø3	-1Ø2	-1Ø1	-1ØØ	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-9Ø	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-8Ø	-79	-78	-77	-76	-75	-74	-73	-72	-71	-7Ø	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-6Ø	-59	-58	-57	-56	-55	-54	-53	-52	-51	-5Ø	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-4Ø	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-3Ø	-29	-28	-27	-26	-25	-24	-23	-22	-21	-2Ø	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-1Ø	-9	-8	-7	-6	-5	-4	-3	-2	-1

BEGINNING Z8Ø MACHINE CODE

LESSON 1

Tim has asked me to write a continuing series on beginning Z8Ø machine code. I should like to hear from you what your specific needs are as well as your suggestions and comments. You can respond direct to me (address at end of article) to avoid delay and extra work for Tim.

We will be developing some machine code (MC) routines to be used as subroutines in a Basic program. This will aid the learning process as well as provide a useful goal. I am laboring under the assumption that you already have a good understanding of Basic. If not, you may wish to master Basic first, as it is a very powerful programing language all by its self.

You may be asking, "What is machine code and why would I be interested in learning it?". Your answer very well may be you are not interested in which case you need read no further. However, if you are tired of your computer taking a coffee break when you ask it to perform some task, you may be interested in how MC can put some speed in your program, and end that coffee break sooner.

What is machine code? Very simply, MC is the 'language' understood by your computers' Central Processing Unit (CPU). Each CPU has its own set of instructions which enable it to perform its task as housekeeper and communicate with the outside world (other devices). These instructions are permanently stored at birth in the CPU as numbers. We will be studying the Z8Ø as it is the CPU Sinclair chose to use in our computers.

I called the CPU, housekeeper, for a reason. The CPU is basicly dumb. It is the 'workhorse' in your computer and can only perform general housekeeping chores. The 'brains' of the operation is the operating system and basic interpreter which is permanently stored in memory (ROM). Without the operating system the CPU is even more helpless than a newborn child. The CPU can only perform as instructed however, it will perform exactly as instructed which may not always be what is intended. More on this later.

We need to pause a moment to define ROM, RAM, Peek and Poke.

ROM and RAM are types of memory locations within your computer. ROM stands for Read Only Memory and RAM for Random Access Memory, niether of which explains their functions very well. All memory locations (addresses) can be thought of storage boxes. All memory may be examined ,however some of these boxes (ROM) cannot be altered while others (RAM) can be. This difference is all we will be concerned with here which brings us to Peek and Poke.

Peek allows us to look at the contents of these boxes, as if they had glass tops. Poke is the command that places a number in the boxes (RAM only). Try Pokeing various numbers, no teddy bears or marbles, into a safe address (30,000 will do) and Peeking the location to see if you attain the results you expected. Try other locations if you think there is something magical about 30,000. Safe locations on the TS1000 are 16509-32767 (16K) and on the TS2068 are 26710-65535. Try an address 0-16383 (ROM). Peek it first, then Poke, then Peek again. You cannot change the contents of ROM but, you are welcome to look inside as often as you wish. You can find the proper syntax for Peek and Poke in your Sinclair manual.

I mentioned the instructions being stored in the CPU as numbers, we also will store our MC as numbers since that is all the CPU understands. Your computers' operating system and basic interpreter are written in MC for this reason. For example, the CPU does not understand the basic command 'PRINT'. There is instead a subroutine of MC instructions (stored as numbers) in the basic interpreter of your computer which tell the CPU what steps are necessary to execute your 'PRINT' command.

To see what I mean, enter and run listing 1 on your computer. You are looking at the numbers (MC) stored in first 20 bytes of ROM. Obviously, you thoroughly understand what they mean. Don't worry I don't understand them either but, the CPU does and thats the point. Those unintelligible numbers mean something to the CPU and shortly will mean something to us.

You should notice that all the numbers are between 0 and 255. This is very important. For now, we need only know that 0 to 255 represents the limit of numbers that can be stored in a single byte. The reasons for this will be obvious later, when we discuss binary numbers. Try changing the values in line 10 to look at other locations.

A quick scan of the Sinclair manual indicates how the computer stores numbers greater than 255. It ties two adjacent bytes together by increasing one of them by a factor of 256 (2^8). Storing numbers this way means we can now use the numbers 0 to 65535 (2^16). The number that we increase by the factor of 256 is refered to as the high byte, making the other the low byte. The oddity in this is that the Z80's designers chose to store the low byte first. For example, let's assume bytes 1 and 2 hold our number. The number held by these two bytes is:

Peek (Byte 1) + 256 * Peek (Byte 2)

You have probably seen this formula before and may have wondered what was happening. Many of the systems variables are stored in this manner as well as all addresses. Even addresses less than 255 require two bytes storage. More on this later, also.

One other point, all the numbers we have discussed thus far are in decimal (base 10) which is how most of us count. Sciencetist and philosophers have debated why we use decimal numbers since approximately 1134 BC (before computers) and the general concensus, despite lack of empirical proof, is that it has something to do with our having ten fingers (that has nothing to do with MC however, I thought you trivia buffs would want to know).

In this series I will represent all MC with hexadecimal (base 16) numbers. This is not an effort to confuse you. I prefer using hexadecimal (hex) numbers for several resaons:

1. all numbers Ø to 255 can be represented with two digits
2. larger numbers can then be represented with a muliple of two digits
3. cleaner/easier screen displays
4. less keystokes to enter MC
5. easier to use and understand the logic instructions
6. my MC loader program handles hex numbers only
7. many MC listings are presented in hex numbers
8. personal preference, after all I'm writing this

The fact that your computer only recognizes decimal numbers is not a problem. The computer is very good at conversions of this nature and I prefer to let it do them.

Now that we know what the hieroglyphrics are called, what is hex? It is base 16 numbers. Just as we represent decimal numbers with the digits Ø-9, we also represent the first ten digits of hex numbers with the digits Ø-9. The next six digits are the letters A-F therefore we count in hex:

Ø,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Don't be alarmed if this confused you. I have provided full conversion charts. Practice using these charts. You will eventually be able to add and subtract, even think in hex. You will see that the chart represents all decimal numbers Ø to 255 as ØØ to FF hex. It is important to realize that hex numbers are always two digits (or some muliple of two) long, even though I show them as single digits above. That is only to avoid confusion over what the first sixteen digits are. Many hex numbers are obviously not decimal however when there could be confusion, you should show a 'h' after the hex number (1Øh). There is no 'd' required for decimal as decimal is the default. Some operating systems use another character to represent hex (such as # or $) however, we will use a 'h'. You may find some hex numbers with an odd number of digits. In this case the first digit is a zero. Some assemblers require this notation in order to distinquise a hex number from a label.

I will leave you with hexadecimal loaders for the TS1ØØØ and TS2Ø68. Enter the appropriate one for your computer and save it for use in subsequent lessons. This loader will serve you well while we are hand assembling our MC due to our routines being short. You do not need a full assembler until you are involved writing much larger MC routines or entire MC programs.

The listings are self explanatory. You are prompted for an address in decimal, then begin entering your code in hex. Enter a 's' to stop. Your code is reflected on the screen to aid error spotting. The TS2Ø68 will automatically save your code and clear the loader from memory, leaving your code above Ramtop. Unfortunately, that cannot be easily accomplished on the TS1ØØØ. We will discuss this in greater detail next time.

Listing 1

```
10 For I = Ø to 19
20 Print I; Tab 6; Peek I
30 Next I
```

Hex Loader TS1ØØØ

```
 10 Print "Enter start address in decimal"
 20 Input a
 30 Cls
 40 Print "Code begins at "; a
 50 Print
 60 Let a$ = ""
 70 If a$ = "" then Input a$
 80 If a$ = "s" then Goto 140
 90 Poke a, 16 * Code a$ + Code a$(2) - 476
100 Print a$( to 2);"  ";
110 Let a = a + 1
120 Let a$ = a$(3 to )
130 Goto 70
140 Save "MC"
150 Save "MC"
```

Hex Loader TS2Ø68

```
10 Cls: Input "Enter start address in decimal"; a: Cl
   ear a - 1: Poke 23658,8: Poke 23609,30
20 Let a = Peek 23730 + Peek 23731 * 256 +1: Print "C
   ode begins at "; a: Let x = a
30 Let a$ = ""
40 If a$ = "" then Input a$
50 If a$ = "S" then Goto 70
60 Poke a, 16 * (Code a$(1) - 48 - (7 and a$(1) > "9"
   )) + Code a$(2) - 48 - (7 and a$(2) > "9"): Print
   a$( to 2);"  ";: Let a = a + 1: Let a$ = a$(3 to )
   : Goto 40
70 Let y = a + 1 - x: Save "mc" Code x, y: Save "mc"
   Code x, y: New
```

<END>

Syd Wyncoop
21Ø7 SE 155th
Portland, Ore 97233
(5Ø3)76Ø-7786

Chart 1

Hex/Dec Conversions

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

BEGINNING Z8Ø MACHINE CODE

LESSON 2

Last issue we discussed Hexidecimal (Hex) numbers and I left you with a machine code (MC) hex loader. You should have noticed these are very simple programs. That is to allow you to enter and debug them easily, as well as make whatever changes you desire. Please feel free to change them.

We now need to explore the nature of MC. Since you are already familiar with Basic, I will draw some comparisons. The first difference is that MC does not use program line numbers to tell the CPU (remember him?) in what order to perform tasks. MC instructions are executed in the order in which they occur in memory. Even after a jump (Goto or Gosub), MC continues to execute the instructions sequentially as they are found at the address jumped to.

Secondly, there are about 7ØØ MC instructions for the Z8Ø as opposed to the 7Ø or so available in Sinclair Basic. Don't let this scare you off. All 7ØØ instructions can be placed in about a dozen catagories and are therefore variations on a theme. We will confine each lesson to one of these catagories.

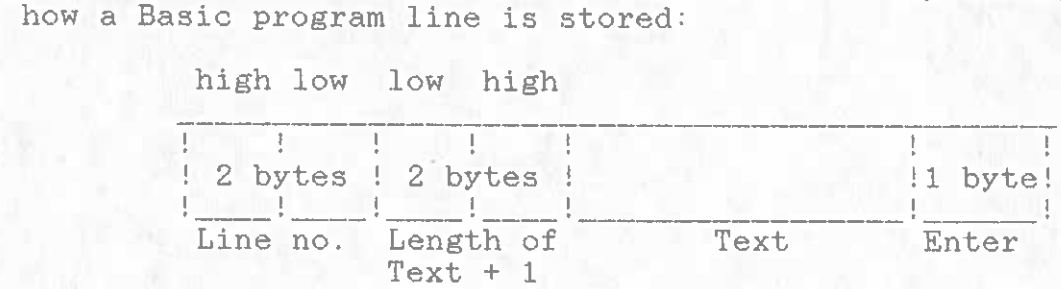
The biggest difference is in how MC 'crashes'. Crash is a term used to describe the condition resulting from an involuntary exit of the program (i.e. stopping with a full screen error, undefined variable error, etc.). When MC crashes there are no error messages to aid us due to the fact that we are not operating within the confines of the Basic interpreter. Often the only recovery is to pull the power plug and begin again. For this reason, I reccomend you always Save your MC prior to execution. Saving it will not prevent a crash but, it does allow for easier recovery.

There are two things to remember that will help prevent MC crashes. First, you cannot use the Break key to stop your MC routine unless it is reading the keyboard and accepting a Break instruction (not likely in most cases). Second, MC will not stop executing unless it is instructed to do so. MC will continue executing instructions (remember all numbers are instructions!) as they are found. The easiest way to solve both problems is to end your routines with a 'return to basic' instruction.

We need to determine where we will store our MC as that is the first prompt in our MC Loader. MC can be stored almost anywhere, although above Ramtop is best, in the TS2Ø68 as it can save bytes as Code. Ramtop is a system variable which tells the Basic operating system how much memory is available and more specifically, what is the last available address in RAM. Ramtop is not necessarily the end of physical memory however, for Basic it is the top of usable memory. Also, all addresses above Ramtop are unaffected by new therefore, your routine cannot be erased. Rand Usr Ø should be used in lieu of unplugging the power lead, if you later wish to clear all memory to the top of physical RAM.

The TS1ØØØ presents special challenges. The best place is still usually above Ramtop however, the TS1ØØØ cannot Save bytes from high memory. We will therefore store our MC in a Rem statement. There are other ways but this is the easiest to Save and execute for now. Later we will find that MC can still be most anywhere.

The ease of execution from the first Rem statement results from our knowing the exact address at which the MC starts. Looking in the Sinclair manual section on memory storage reveals how a Basic program line is stored:



The first two bytes are the line number and note they are in direct oposite order of the normal storage of two byte numbers. The next two bytes are stored as the Z8Ø would normally store numbers and represent the length of the text in the line plus the Enter, which is used by the Basic interpreter as an 'end of line' marker. Next follow the Basic text and finally the EOL marker. This makes the first byte after Rem the sixth byte in the line and in the program area if the line is the first line of the program. This address is 16514 in the TS1ØØØ since the Basic program area begins at 165Ø9. We will insure that we are working with the first line as follows. Type:

1 REM ENTER, POKE 1651Ø,Ø ENTER

We have also insured that our first line cannot be Edited even though it will still Save.

The next thing we need to do is make space in our Rem statement to hold our MC. Refering to the chart above, the EOL marker is next after Rem, since we have not entered any text. We must never overwrite the EOL marker as we will cause an awful nasty crash. Type four lines of spaces after the Rem (you can figure out how to edit it) and Save your MC loader with your line Ø to avoid retyping it next time. This is very wasteful of memory but, will serve us well for now. Your Rem statement need only contain the exact number of bytes you need when working within a program.

We now need to know how to execute (Run) our MC. This is accomplished with the USR function. The Sinclair manual is a little vague on its use. The proper syntax is:

Command USR X

Where; Command=most Basic commands
USR=USR function
X=address to begin executing from
Examples; RAND USR 16514
PRINT USR 16514
LET A=USR 16514

Boy, this is sure good stuff but, I ain't written no MC program yet! Well hang in there, we will get to the actual instructions next issue. Right now though how about a sneak preview?

You may have heard of such terms as opcodes, mnemonics assembly and disassembly. Opcode is short for Operation Code and is the Hex numbers we will be entering. They could just as easily be represented in decimal or binary however, we have chosen hex.

Mnemonics are another shorthand which has been designed especially for us humans. The CPU understands a long list of numbers (opcodes) however I don't. I do understand mnemonics as they are almost english (I did say almost). Look at the sample disassembly below to see what I mean.

Assembly is what we will be doing when we convert our MC programs to hex. We will be 'hand assembling' our MC. Assembly Language is another term for MC and is usually used to refer to the Opcodes.

Disassembly is the opposite of assembly and is usually used to refer to a 'listing' of MC instructions. You will probably want to disassemble someone elses MC after you understand what the mneumonics mean. That can help your understanding and learning of MC as you will already know what the program does and will be able to see how the task at hand was accomplished. As in Basic, there is no single best way to program in MC. We all develope our own style (or lack of it).

I will end this lesson with a sample disassembly. May we soon know what it means.

Address	Label	Opcodes	Mnemonics	Comments
16514	start	3EØA	Ld A,ØAh	;Put ØAh in A register
16516		Ø61Ø	Ld B,1Øh	;Put 1Øh in B register
16518		8Ø	Add A,B	;Add ØAh & 1Øh and place
16519		4F	Ld C,A	;result in BC register
1652Ø		Ø6ØØ	Ld B,Ø	
16522	done	C9	Ret	;Return to Basic

<END>

Syd Wyncoop
21Ø7 SE 155th
Portland, Ore 97233
(5Ø3) 76Ø-7786

BEGINNING Z8Ø MACHINE CODE

LESSON 3

Before we get to our first MC instructions, lets take another look inside our CPU. Inside we will find registers that are called A,F,B,C,D,E,H,L,I,R,IX,IY,SP,PC,A',F',B',C',D',E',H' & L'. These are not the alphabet soup CPU had for lunch. Registers are merely storage places within the CPU as opposed to external memory (ROM & RAM). Think of these registers as storage boxes with names instead of addresses, much the same as you would Basic variables.

Some of the single registers can be married to form register pairs. You are hereby ordained, by the power invested in me by the Great God Z8Ø, as Justice of the Cpu, to form these unions as required. The permissable combinations are AF,BC,DE,HL, AF',BC',DE' & HL' (and you thought I didn't know the alphabet!).

Single registers are similar to bytes in that they can contain any value Ø-255. Register pairs can contain any value Ø-65535 which makes them very valuable as address pointers. Refer to the discussion on addresses in lesson 1 for more on this. (contact TDM if you need back copies). We will use these similarities to pass parameters (information) to and from our MC routines.

On the subject of addresses and register pairs, you need to remember which is high and low. In memory (addresses) the first byte is low however, with register pairs the first register is high. This is easily remembered by knowing that the HL register pair was named with this in mind. H means high and L means low. An assembler will handle this for you but, we will have to watch it while we are hand assembling our code. Many crashes will occur because you forgot or confused the order of the high and low bytes or registers.

Some of these registers have special names and/or jobs. Chart 2 lists some of these names/jobs. However, we will not discuss them further until we get to the instructions using them.

Now for our first set of instructions (and you were wondering if I even knew any). Its mnemonic is Ld, which is short for Load. Ld has no relation to the Basic LOAD command. Ld is an assignment instruction and acts very much like the Basic LET command.

The proper 'syntax' is Ld A,15. Which is read as 'Load the A register with the value 15'. Ld acts very much like the Basic LET x=15.

Take another look at the sample disassembly I left you with last lesson. Look at the comments and see if you can follow what is happening. It is a program that will return the sum of ØAh and 1Øh to basic with the command: PRINT USR address. For practice, you can enter that program. Try poking the 2nd and 4th bytes with different values and run it again to see if you get the results you expect. If the sum is greater than 255 you will discover a bug I left for later correction.

Note that we loaded the result into the BC register pair before returning to basic. This is due to the Basic operating systems' handling of the USR function. It will always return the value held in the BC register pair. The value returned will not be the result unless you properly load BC before returning.

Ld may not seem to be of much value however, in its many forms, Ld is the most used instruction. We can Ld most registers, register pairs or addresses with either a constant, the contents of another register(pair) or the contents of an address. Chart 3 details some of the many forms Ld can have as well as the proper 'syntax'.

You will notice some instructions have parenthesis. The parenthesis signify 'the contents of'. For example; read the instruction Ld A,(4000h) as Load the A register with the contents of the address 4000h. The Basic commands PEEK and POKE can be compared to these instructions. If the parenthesis appear on the left of the comma, you have a POKE operation and if they appear on the right of the comma, you have a PEEK operation. The Basic equivalent of Ld A, (4000h) is LET x = PEEK 16384, (4000h=16384). Using this knowledge, the instruction Ld (4000h), A is equivalent to POKE 16384,x.

You will also notice a symmetry to the instructions. You can Ld r, (HL) and you can Ld (HL), r. This symetry will prove to be very useful and holds true throughout most of the instruction set.

Note that some instructions seem to favor the register A or the register pair HL. This is due to their special functions (chart 2). There are simply some instructions that can only be performed with A or HL and no other register (pair). We will see that Ld is not the only instruction to exhibit this favoritism. This is not as restrictive as it first sounds, although you will on occasion wish for an instruction that does not exist.

There is no need to detail the operation of each instruction as you should be able to determine approximately what can be expected from them, if you study charts 2 & 3 in conjunction with this lesson. We will discuss them further as we use them. It will be much easier for me to explain and you to understand their operation then.

I am not listing the hex codes for all the Z80 instructions we will use as this is not intended to be an exhaustive study but, is meant to give you a start. The first rung of the ladder. If you have not yet obtained a good book on the subject, you can find the codes in the appendix of your Sinclair manual.

I cannot hope to give you all you will need in an article such as this. I must advise you to get a good book as a study guide and to fill in where I leave off. Rather than suggest a book that you may not like as well as I do, I would advise you to look at several. If possible, get several opinions but get a book.

That's it for now. Next issue we will discover the math instructions. The special significance of A and HL will be very obvious after that.

<END>

Chart 2

Register	Name	Job
A	Accumulator	accumulate the results of eight bit arithmetic directly access the contents of any memory address
F	Flags	holds various flags for CPU which indicate the results of arithmetic and logical instructions
B		eight bit counter
BC		sixteen bit counter
DE	Destination	used for block moves
HL	High/Low	sixteen bit arithmetic directly access memory addresses indirect address pointer

Chart 3

Registers	Register Pairs
Ld r,r	Ld rr,nn
Ld r,n	Ld IX,nn
	Ld IY,nn
Ld A,(pq)	
Ld (pq),A	Ld (pq),BC
	Ld (pq),DE
Ld r,(HL)	Ld (pq),HL
Ld A,(BC)	Ld (pq),IX
Ld A,(DE)	Ld (pq),IY
Ld (HL),r	
Ld (BC),A	Ld BC,(pq)
Ld (DE),A	Ld DE,(pq)
	Ld HL,(pq)
Ld r,(IX+d)	Ld IX,(pq)
Ld r,(IY+d)	Ld IY,(pq)
Ld (IX+d),r	
Ld (IY+d),r	
Ld (HL),n	
Ld (IX+d),n	
Ld (IY+d),n	

Where: r =any single register
rr=any register pair
n =any numeric constant Ø-255
nn=any numeric constant Ø-65535
d =any displacement Ø-255
pq=any address Ø-65535

BEGINNING Z8Ø MACHINE CODE

LESSON 4

This time, right to business! We are studying the math instructions which are listed in chart 4. This is where it starts getting a little more difficult but, not so that you cannot handle it. Up to this point most of the lessons have been periferal background needed to make sense out of the rest of the discussion.

We only have two math functions available to us, Addition and Subtraction. As with Ld, this is not as limited as it first sounds. A study of Math Theory would teach you that all math functions are performed with addition. I'll not try to explain this further as it would fill a volume larger than all the TDM's published to date. The point we need to understand and absorb is that multiplication is performed by repetitive additions. Likewise, division can be achieved by repetitive subtraction.

It is important that this make sense to you. Think about the multiplication problem of 12X6. It can be solved by either of the following:

```
  12          12
x 6           12
 72           12
              12
              12
             +12
              72
```

Can you see how we can solve division problems by repetitive subtraction? If we had the problem 72/12, how many times can we subtract 12 from 72? Is there a remainder? Simple, isn't yet?

This brings us to the first instruction, Add. We have already seen Add in operation, in Lesson 2, and probably have a good idea of its function. Trust me, it performs addition. Some of the later instructions will not be so obvious. We would read the instruction, Add A,E, as 'add the value in the E register to the value in the A register and store the result in the A register'.

In lesson 3, we learned the A register is called the Accumulator. The A register is the only register that can directly 'accumulate' the results of eight bit arithmetic. If we had wanted the result in the E register, we would need to assign it. Can you guess the needed instruction? You get an A if you said Ld E,A. Otherwise, go back to lesson 3.

We also have available the instruction, Sub. The A register performs a special purpose here also. The A register is the only register we can subtract from. As with Add, the A register accumulates the result. You may see this instruction written as Sub A,C or Sub C. They mean the same thing. We will use Sub C as the A register is always implied in eight bit arithmetic.

I have mentioned several times that the A register will accumulate the results of eight bit arithmetic. We need to leave the instructions for some more background.

We have already learned that a single register may only contain a value in the range Ø-255. There is a condition, known as an 'overflow' which occurs when these values are exceeded. The simplest way to describe overflow is by example. Let's assume we are adding 255+1. We have not discussed number systems yet (that's a later lesson) but let's show our example in binary as it will demonstrate the point dramatically:

Decimal	Binary
255	11111111
+ 1	+ 1
256	1 ØØØØØØØØ

Look closely at the binary example. Each digit represents a bit of the A register (or any other eight bit location). Assume for now that my answer is correct and you will note that we are now trying to place a nine bit number into an eight bit hole! The answer returned in this case would be Ø instead of the expected answer of 256. Our example shows an eight bit overflow but, can you see how we can also overflow a register pair? (sixteen bits).

Our friend, CPU, has a special register, F, which we learned stands for Flag. It is called this because its job is to keep track of various things for CPU. This is accomplished by the setting or resetting of a bit of the F register. Setting a bit makes it a 1 and resetting it makes it a Ø. We will discuss this in some detail at a later time.

The bits are referred to as flags due to the fact that they can indicate whether or not a certain condition exists. The flag we are interested in now is the Carry flag. We will also discuss the F register later therefore, we only need consider the carry flag now.

In the above example, we found we would receive an answer of Ø. The ninth digit is not lost, as it is placed in the F register as the carry flag. In other words, the carry flag takes on the value (either 1 or Ø) of the overflow from our arithmetic operation. We will soon see why we would want to save the carry.

Back to the math instructions. We have available the instruction ADC which is read add with carry. To see the difference, another example:

```
Add A,E   means   Let A=A+E
ADC A,E   means   Let A=A+E+Carry (keeping in
                  mind that the carry will again
                  be set or reset by the result)
```

ADC will allow us to chain together the needed additions to guarantee the correct result. Some of the same results can be achieved with the register pair instructions however, there can still be overflows. Study the following to see what I mean:

```
Ld HL,ØØ4Øh          Ld H,ØØh
Ld BC,7FFFh          Ld L,4Øh
Add HL,BC            Ld B,FFh
Ld B,H               Ld C,7fh
Ld C,L               Ld A,L
Ret                  Add A,C
                     Ld L,A
                     Ld A,H
                     ADC A,B
                     Ld B,A
                     Ld C,L
                     Ret
```

Both of these routines will do the same job. Which makes more sense? Uses less memory? Executes faster? The point is that there are many ways to get the job done and many considerations to why we should choose one over another.

We also have SBC or subtract with carry. This one is special because it is the only way to perform sixteen bit subtraction. We cannot Sub HL,BC. We must SBC HL,BC which implies we know the status of the carry flag. We may not know what's on carry's mind but we can clear the carry flag prior to performing a SBC by doing an addition that we know will not generate a carry. One that will work in all cases is Add A,Ø. The value of A is unchanged and the carry flag is reset (Ø) or cleared as there is no overflow. We will find other ways to clear carry, soon.

We need to be aware that HL acts as the accumulator for sixteen bit arithmetic operations. HL has much the same favorite status with CPU as does A. The reason we need an eight and a sixteen bit accumulator is that we cannot add or subtract registers from register pairs and vice versa. In other words, we cannot Add HL,A. It should be obvious that a sixteen bit result will not fit in A, however the reverse is not quite as obvious. If we wish to Add HL,A, what is the high byte? It simply will not work.

The last instructions for this lesson are special cases of Add and Sub. They are Inc and Dec which are short for increment and decrement. Each will Inc or Dec by one. For example:

```
Inc HL     means    Let HL=HL+1
Dec HL     means    Let HL=HL-1
```

Armed with these new instructions, see if you can rewrite the addition routine we had in Lesson 2, to avoid the overflow error it contains. Make sure the last instruction is a Ret and use PRINT USR address to run it and return the answer to basic. See if you can write a similar routine to perform subtraction.

A final note on the charts I am providing. This is the last time I will include the abbreviations comments. Also, you can usually substitute IX or IY for HL and (IX+d) or (IY+d) for (HL) therefore, I will not include them in the charts.

Until next time, happy computin'.

<END>

Syd Wyncoop
21Ø7 SE 155th
Portland, Ore 97233
(5Ø3)76Ø-7786

Chart 4

Registers	Register Pairs
Add A,r	Add HL,rr
Add A,n	Add HL,SP
Add A,(HL)	Add IX,rr
Add A,(IX+d)	Add IX,SP
Add A,(IY+d)	Add IY,rr
	Add IY,SP
ADC A,r	
ADC A,n	ADC HL,rr
ADC A,(HL)	ADC HL,SP
Sub r	SBC HL,rr
Sub n	SBC HL,SP
Sub (HL)	
	Inc rr
SBC A,r	Inc SP
SBC A,n	
SBC A,(HL)	Dec rr
	Dec Sp
Inc r	
Inc (HL)	
Dec r	
Dec (HL)	

Where: r =any single register
rr=any register pair
n =any numeric constant Ø-255
nn=any numeric constant Ø-65535
d =any displacement Ø-255
pq=any address

BEGINNING Z8Ø MACHINE CODE

LESSON 5

I left the last lesson with a challenge to you to rewrite the sample disassembly from lesson 2 to eliminate the overflow error it contained. If you had difficulty, refer to lesson 4. The answer was given in the comparison which explained the ADC instruction. How many of you thought of rewriting the routine using the sixteen bit instructions? Did you use LD HL,(pq) and LD BC,(pq)? Can you see how a short Basic interface (program) could collect the values and call the MC routine to perform the addition? I trust some of you are beginning to have some ideas.

We now know how to load a register (pair) or memory location and perform arithmetic with the values loaded. We would however, find MC of very limited value if these were all it could do. Most of you are familiar with the Basic commands GOTO and GOSUB. In truth, it is these instructions that give a program the power to do some real work for us.

In MC, the equivalent instructions are referred to as Jumps and Calls. The syntax for these instructions is given in chart 5. You will note a new abbreviation, c, which is a test for the condition (or status) of a flag.

We briefly discussed the Carry flag last lesson. Here is how the F (flag) register is arranged:

Bit#	7	6	5	4	3	2	1	Ø
Flag	S	Z	.	H	.	P/V	N	C

Where:
- S = Sign
- Z = Zero
- H = Half-Carry
- P/V= Parity/Overflow
- N = Subtract
- C = Carry
- . = Not used

Sign Flag - Stores the sign of the last result. Flag will be set for a negative result and reset for a positive result (always reflects the most signifigant bit of the result).

Zero Flag - Checks whether last result was zero. Flag will be set if result is zero, else reset. Note: flag = 1 if result = Ø. Watch it!

Half-Carry- Used internally by CPU to record carry from bit 3 to bit 4 in registers or bit 11 to bit 12 in register pairs. We will ignore it.

Parity/Overflow- Has two jobs depending on the instruction last executed.
Parity is the number of set bits in the result

and is referred to as odd or even. Flag will be set if parity is even and reset if odd. Note: even parity generates an odd flag. Watch this one, also!
Overflow records a carry from bit 6 into bit 7 which effectively changes the sign or result in signed arithmetic operations. Flag will be set for overflow, else reset.

Subtract Flag- Used internally by CPU to record whether last instruction was addition or subtraction. Flag will be set if was subtraction operation. We will ignore this one, also.

Carry Flag- Our old friend records a carry from bit 7 to bit 8 in registers or bit 15 to bit 16 in register pairs. Is also used to save the lost bit in the shift and rotate instructions.

You will note that two bits of the flag register are unused. The status of these bits is unimportant and there are no instructions that affect them.

Each flag can be in one of two states, set or reset (on or off). A set bit = 1 (on) and a reset bit = Ø (off). This can become very confusing when using the Zero or Parity/Overflow flags as the flag will not be as we expect it. For instance, the Zero flag = Ø if the result was not zero. Most of the time however, you can use the flags without knowing whether they are set or not. You need only test their status and jump accordingly.

Each flag indicates a specific condition based on the result of the last instruction executed. Chart 6 indicates how the flags are affected by the various instructions. It is important to know how the flags are affected as every instruction does not affect them and many instructions do not affect them as you might expect.

Enough of that, back to the Jump instructions. This instruction has two versions, Jump and Jump Relative. The mnemonics are JP and JR, respectively.

JP is equivalent to Basics' GOTO. JP begins executing the next instruction at the absolute address you specify as its argument. A JP 4ØØØh instruction will send CPU off to address 4ØØØh to find the next instruction to execute. Your jumps can be conditional, that is, they can test one of the flags and jump only if the condition is met.

JR requires the introduction of another Hex to Decimal conversion chart, Chart 7. You will note that the first half of this chart is the same as our previous Hex to Dec chart (lesson 1). The last half however indicates negative numbers. When numbers are used in this fashion, they are referred to 'signed numbers'. Signed numbers merely means that the most significant bit (bit 7) is used to represent the sign of the number. A set bit (1) is a negative number and a reset bit (Ø) is positive.

JR also requires a brief discussion of the register pair PC. PC is a special register pair not normally accessible to us. It is called the Program Counter and its job is to keep track of where the next instruction to execute is located. All Z8Ø instructions are 1,2,3 or 4 bytes in length. CPU will always advance PC by the correct number of bytes for the instruction it

is about to execute. The effect of this is to skip any arguments belonging to the current instruction so as to be in position to fetch the next instruction.

Any jump instruction causes PC to discard the address it contains and replace it with the new address, as specified in the jump instruction. Note, PC will always contain the address of the next instruction to execute, not the current one.

The JR instruction adjusts the PC by adding the value specified to the current value of PC. In other words, JR tells the CPU to Jump to address X, which is Y bytes from where PC is. Y can only be in the range of -128 to 127 and X is the calculated new address. In the case of negative values, the program would jump back to a previous instruction (loops) while positive numbers would cause the skipping over of the next Y bytes.

JR can also be conditional as indicated in chart 5 and discussed above for JP.

When programming in Basic, it is quite common to have a line such as:

```
10Ø GOTO 1Ø*VAL A$+1ØØØ
```

There is a MC instruction, JP (HL), which emulates this type of operation. This instruction will jump to the address held in the HL register pair. This allows a routine to build up an address from tables or inputs and transfer program control to that address. We will not discuss this much further now as it represents some pretty advanced programming.

CALL is our GOSUB equivalent. It acts exactly like Basics' GOSUB. A jump is made to the specified address and a return is made to the instruction that would have been executed next had the CALL not been encountered. This is accomplished by saving the address in PC on the stack (we will explain the stack later) before making the jump.

There is a special case of CALL that does not require an address to be specified, which is known as RST. RST is read restart and is unique because it is the only instruction that uses an eight bit address. RST calls a subroutine with a one byte instruction.

Some important points about RST are that it is unconditional and usually computer specfic (can not run on another Z8Ø based computer). Being computer specific is due, unfortunately, to there already being instructions at all the RST addresses, which cannot be changed. This is due to our operating system being in a ROM type of memory. All is not lost though. Since these are very handy instructions, Sinclair put some of the most accessed routines there. We will find that we can use some of the RST instructions, after all.

As with any GOSUB instruction, Calls and RSTs require a return instruction to let the CPU know the routine has finished its task. The mnemonic for return is amazingly enough RET. RET will perform exactly the operation you would expect it to and your returns can be conditional. Conditional returns allow for many exit points based on completing certain tasks. There are two special RETs which we will discuss later because they are used to return from the interrupts.

We have learned about the flags and how to make jumps and calls based on their status. We now need to explore some of the ways to set these flags in order for our tests to be meaningful. One of the ways to do this is directly with the CCF and SCF

instructions.

CCF means Complement the Carry Flag. If Carry was set, it will be reset and vice versa.

SCF means Set the Carry Flag. The Carry flag will set by this instruction.

Another way to affect the flags is with the remainder of the arithmetic instructions (I've been holding out on you again). These are also listed on chart 5 and can not truely be referred to as arithmetic instructions, except for CP.

CP, which means compare, is a neat and often used instruction. CP sets all the flags as if a value were subtracted from the Accumulator but, without changing the value of the accumulator! It is important to realize the result of the compare is not stored anywhere, only the flags are affected.

CP has two special forms, CPI and CPD, which are read Compare with Increment and Compare with Decrement. CPI performs the same as a CP (HL) instruction would except that HL is incremented and BC is decremented. The only flag affected is the P/V flag which is set according to the value of BC. If BC = Ø, then P/V = Ø.

CPD is the same as CPI except that HL is decremented. the effect on the flags is the same.

The next instruction is DJNZ which is not Greek. DJNZ is read 'decrement the B register and jump relative if B is not zero'. This is an extremely useful instruction which leads to the B register being used as a counter. DJNZ can be compared to the Basic loop control variable. The equivalent Basic statement would be as follows:

```
1Ø For X = 1Ø to Ø Step -1
2Ø (do job here)
3Ø Next X
```

In order to perform the same operation as DJNZ using any other register, you would need two instructions:

```
DEC L
JR NZ, Loop
```

To use DJNZ, you must properly load the B register. You can then construct a loop to do whatever task you wish. You can even reuse the B register in the loop, if you properly preserve its value first. More on this preservation of values later.

CPL stands for complement. Each bit of the Accumulator is altered (complemented). For example: if the accumulator contains 11Ø111Ø1b, its complemented form would be ØØ1ØØØ1Øb.

NEG is the last unexplained instruction on chart 5. NEG will negate the accumulator, which means to place the twos' complement of the A register in the Accumulator. If the accumulator contains 5, it will be negated to -5.

You now have about one third of the Z8Ø instruction set and, with the stack instructions next issue, they are certainly the most used of the instructions. You are now armed with the tools to write a MC program of your own design. I encourage you to experiment and see if you get the desired results. I will reply personally to all enquiries that contain a self-addressed, stamped envelope, if you have difficulty.

With the next lesson we will explore printing to the screen

as that will give us some immediate feedback as to how we are doing and whether our routine is working. If you have any information on the display file and/or ROM routines, you should review it, in anxious anticipation.

<END>

Chart 5

Jumps	! Flag setting
JP nn	! CCF
JP c,nn	! SCF
JP (HL)	!
	! CP n
JR e	! CP r
JR c,e	! CP (HL)
DJNZ e	! CPI
	! CPD
CALL nn	!
CALL c,nn	! CPL
RST xx	!
	! NEG
RET	!
RET c	!

Where: n = any numeric constant Ø to 255
nn = any numeric constant Ø to 65535
r = any single register
e = any numeric constant -128 to 127
c = flag status
xx = ØØh, Ø8h, 1Øh, 18h, 2Øh, 28h, 3Øh, or 38h

Chart 6

Instruction(s)	C	Z	P/V	S	N	H	Comments
ADD, ADC	*	*	V	*	Ø	*	8 bit add or add w/carry
ADD	*	-	-	-	Ø	-	16 bit add
ADC	*	*	V	*	Ø	-	16 bit add w/carry
AND	Ø	*	P	*	Ø	1	Logical operations
BIT	-	*	-	-	Ø	1	Specified bit copied into zero the flag
RES, & SET	-	-	-	-	-	-	Bit instructions
CCF	*	-	-	-	Ø	-	Complement carry flag
SCF	*	-	-	-	Ø	Ø	Set carry flag
CP, NEG, SUB, SBC, DEC, & INC	*	*	V	*	1	*	8 bit subtract or subtract w/carry, compare or negate accumulator & 8 bit decrement
DEC, & INC	-	-	-	-	-	-	16 bit decrement and increment
SBC	*	*	V	*	1	-	16 bit subtract w/carry
CPI, CPIR, CPD, & CPDR	-	*	P	-	1	-	Block searches; Z=1 if A=(HL), else Z=Ø;P/V=1 if BC not equal to Ø, else P/V=Ø
CPL	-	-	-	-	1	1	Complement accumulator
DAA	*	*	P	*	-	*	Decimal adjust accum.
IN	-	-	-	-	-	-	Input register direct
IN	-	*	P	*	Ø	Ø	Input register indirect
INI, IND, OUTI, OUTD, INIR, INDR OTIR, & OTDR	-	*	-	-	1	-	Block in & out instructions; Z=Ø if B is not equal to Ø, else Z=1
LD	-	-	-	-	-	-	Assignment instructions
LDI, LDD, LDIR, & LDDR	-	-	P	-	Ø	Ø	Block transfers; P/V=1 if BC is not equal to Ø else P/V=Ø
OR, & XOR	Ø	*	P	*	Ø	Ø	Logical OR accumulator
RLA, RLCA, RRA, & RRCA	*	-	-	-	Ø	Ø	Rotate accumulator
RL, RLC, RR, RRC, SLA, SRA, SRL	*	*	P	*	Ø	Ø	Rotate and shift left or right

Where: * = Flag changed according to result
- = Flag either unchanged or undeterminable
Ø = Flag reset
1 = Flag set
P = Parity changed according to result
V = Overflow changed according to result

Chart 7

Signed Numbers--Hex/Dec Conversions

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	-128	-127	-126	-125	-124	-123	-122	-121	-120	-119	-118	-117	-116	-115	-114	-113
9	-112	-111	-110	-109	-108	-107	-106	-105	-104	-103	-102	-101	-100	-99	-98	-97
A	-96	-95	-94	-93	-92	-91	-90	-89	-88	-87	-86	-85	-84	-83	-82	-81
B	-80	-79	-78	-77	-76	-75	-74	-73	-72	-71	-70	-69	-68	-67	-66	-65
C	-64	-63	-62	-61	-60	-59	-58	-57	-56	-55	-54	-53	-52	-51	-50	-49
D	-48	-47	-46	-45	-44	-43	-42	-41	-40	-39	-38	-37	-36	-35	-34	-33
E	-32	-31	-30	-29	-28	-27	-26	-25	-24	-23	-22	-21	-20	-19	-18	-17
F	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

BEGINNING Z8Ø MACHINE CODE

LESSON 6

This lesson we will discuss the stack and the instructions which use the stack. What is a stack? A stack is simply an area of consecutive bytes of memory which are used for storage by the CPU. The CPU cannot operate without a stack. We will find that we too can use the stack, if we are careful.

Remember our earlier discussion of Rom, Ram and boxes? If not, you need some back issues! Think of our CPUs' stack as a stack of boxes (memory locations). You can remove or add to the top of the stack easily but, try to remove or add a box somewhere in the middle and the stack topples. CPUs' stack works the same way except it grows down from the top as if the boxes were suspended from the ceiling. Therefore, we actually add to the bottom of the stack.

There is a special register inside CPU dedicated to keeping track of the stack. Its mnemonic is SP which means Stack Pointer. SP contains the address of the last location on the stack.

All information on the stack is stored in the usual two byte format used for addresses. We can place information on the stack (PUSH) or remove it from the stack (POP). Our friend CPU automatically adjusts the SP with each operation by the required two bytes. It is important to realize that even though SP is adjusted to point to the correct location (box), the information is still there until it is overwritten. See figure 1 to make this clearer.

The PUSH and POP instructions can add/remove information to/from the stack and any register pair. For instance, if we wish to stack the contents of the B register, we need to PUSH BC. We will have also stacked the C register since, we must use a register pair.

Last issue we learned the CALL instruction. It uses the stack to save the value of PC in order to know where to return to. In effect, CALL executes a PUSH PC, JP to new location and complete the subroutine, and then a POP PC (Ret) and continue executing the program from the byte after the CALL instruction.

The next instruction is Ld SP,HL. This is a simple assignment instruction. Whatever value is held in HL will be copied into SP, not the stack. Remember, most instructions assume all values on the stack to be valid addresses, even if they are data, so it is important to know where SP is.

The last instruction affecting the stack is EX (SP),HL. It will exchange the contents of the address referenced by the SP with the value held in HL. Assume HL = 1Ø4Øh and SP = 9Ø5Øh and address 9Ø5Øh = 59 and address 9Ø51h = 68. After the EX (SP),HL instruction is executed, their new values will be; HL = 6859h, SP = 9Ø5Øh, address 9Ø5Øh = 4Øh and address 9Ø51h = 1Øh. Notice that SP is unaffected however, the contents of the last stack entry are changed.

This is a good time to introduce the other exchange instructions. They are all fairly easily understood and are

listed in the chart. Note that an exchange merely swaps the contents of the affected registers and no others, neither are any flags affected, except for the EX AF,AF' instruction. The EX AF,AF' exchanges only these registers while the more general EXX exchanges BC, DE & HL with BC', DE' & HL'.

These exchange instructions do not actually change the register contents. Consider the EX AF,AF' instruction. The AF register becomes the AF' register and the previous AF' register pair becomes the new AF register pair. This is important as the contents of registers can be stored out of the way for later retrieval. It also means we must be sure of which set of registers are in use.

The EX DE,HL instruction is very useful and will exchange the contents of DE with the contents of HL. This is the same as if there had been an instruction to Ld DE,HL and Ld HL,DE without disturbing any of the values. a series of PUSHes and POPs would be needed to accomplish the same result. For example, lets EX BC,HL (there is no such instruction):

```
PUSH BC
PUSH HL
POP BC
POP HL
```

Note that the information was moved from the stack into a different register pair than it originated from. This is a very useful tool to have at our disposal however, we must be aware of what we are doing or we may find ourselves expecting data at a location other than where it ended up.

You will no doubt have noticed that I have rather laboriously explained the many instructions we have learned up to this point. The truth of the matter is that I have been trying to walk the fine line of too much detail/not enough detail. I hope there has been enough enough to get you started without boring anyone.

We now have enough instructions to begin programming. I firmly believe the only way to learn any language is to use it. With that in mind we will concentrate more on accomplishing some task and less on the instructions. I must assume that if you are still with me, you have by now acquired some good books to supplement your learning.

We will need to be able to 'see' if our programs are completing the task as we desire therefore, we will initially write programs that will affect the display file. This will necessitate two separate discussions as the TS1000 and TS2068 each handle their display files differently. You may wish to skip the section which does not pertain to your computer but, I think you will find it beneficial to read.

Before we jump right into it though, we need to take a look at the Sinclair manual again. Towards the back of the manual you will find a section on the systems variables. These are variables used by the Basic operating system to keep track of various items. Many of these will prove useful to us and several others are required to be under our full control. I will use Sinclairs' names and explain each one as we need it. You should take a moment to review this section of the manual as we will become intimate with many of the system variables.

The display file (D-File) is arranged as 24 rows of 33 characters. The last character in each row is an end-of-line (EOL) marker, which is Chr$ 118 (the code for Enter). In addition the very first character is an EOL marker. We must never ever, never overwrite any of the EOL markers. If we overwrite any of the EOL markers, the system will crash!

This description only applies to a fully expanded system (greater than 3.25K). The D-File is collapsed to 25 EOL markers in a smaller system. I will assume yours is fully expanded.

Since the D-File moves about in memory as your Basic program expands and contracts, its location is held in a system variable known as none other than D-File. This means that we can always locate the D-File with the instruction Ld HL,(D-File).

The easiest way to print to the D-File is to use the RST 1Øh instruction as that is where Sinclair has placed the print routine. RST 1Øh will print whatever character is in the A register. Enter the following to get a full screen of asterisks:

Listing 1

```
ØE16     Start   Ld C,18h          ;line counter
Ø62Ø     Loop1   Ld B,2Øh          ;characters/line counter
3E17     Loop2   Ld A,17h          ;character to print
D7               Rst 1Øh           ;go print it
1ØFB             DJNZ,Loop2        ;until line is full
ØD               Dec C             ;count one line done
2ØF6             Jr NZ,Loop1       ;another line?
C9       Done    Ret               ;return to basic
```

While RST 1Ø is the easy way, it is only a minor improvement over Basic. That's because we are using the same routine as Basic uses. The advantage is that we didn't have to keep track of the EOL markers.

The fastest way to print to the screen is by direct pokes, even from Basic. Enter the following for an almost instant screen fill:

Listing 2

```
2AØC4Ø   Start   Ld HL,(D-File)   ;get D-File location
ØE18             Ld C,18h         ;line counter
23       Loop1   Inc HL           ;get past EOL
Ø62Ø             Ld B,2Øh         ;characters/line counter
3617     Loop2   Ld (HL),17h      ;poke character onto screen
23               Inc HL           ;advance print position
1ØFB             DJNZ,Loop2       ;go do again?
ØD               Dec C            ;count one line done
2ØF5             Jr NZ,Loop1      ;do another line?
C9       Done    Ret              ;return to Basic
```

You should have noticed that this method allowed printing on all lines. There is a system variable, DF_SZ, which can be

poked from MC or Basic to allow full screen printing however, the system can be easily crashed if not properly handled. Also, the number 17h can be any print-able character code.

Now for an all purpose, generic print routine:

Listing 3

```
1A      Print   Ld A,(DE)        ;check for EOL marker
FE76            Cp 76h
2001            Jr NZ,NoEOL
13              Inc DE           ;get past EOL marker
7E      NoEOL   Ld A,(HL)        ;get character to print
FEFF            Cp FFh           ;check for end of text
C8      Exit    Ret Z            ;and return if reached
12              Ld (DE),A        ;print it
23              Inc HL           ;advance character pointer
13              Inc DE           ;advance print position
18F3            Jr Print         ;do it again
```

The print routine is useless by itself. Upon entry, HL must contain the address of the first character to print and DE must contain the address in the D-File to print at. Enter the following routine to understand how you would set-up HL & DE and Call this routine at Print.

Listing 4

```
2A0C40  Entry   Ld HL,(D-File)   ;get D-File location
EB              EX DE,HL         ;into DE
210041          Ld HL,Text       ;get address of text
                                 ;address assumed to be 4100h
CD8240          Call Print       ;go print message
                                 ;address assumed to be 4082h
C9      Done    Ret              ;return to basic
```

And now a message must be stored at 4100h. Enter these hex codes to address 4100h:

Hex Dump 1

```
39 2D 2E 38 00 2E 38 00 26 00 39 2A 38 39 1B
00 2E 00 38 3A 37 2A 00 2D 34 35 2A 00 2E 39
00 00 3C 34 37 30 38 1B FF
```

Note that the Print routine requires a terminating byte FFh in order to exit. Our test should now work with the command Rand Usr Entry.

****

The D-File consists of 192 lines of 32 bytes/line for the character information and 24 lines of 32 bytes/line for the attribute information. The last 768 bytes are known as the Attributes File (A-File). The D-File is fixed in memory at address 4ØØØh and the A-File resides at 58ØØh. We will discuss the A-File at another time as its purpose is to hold the color attributes of each character square. We will therefore not be using the A-File at this time.

The organization of the D-File is not what you would expect. Each character is eight pixels by eight pixels (one character square). The eight pixels across fit nicely in one byte, hence the 32 bytes across each line. The problem is the eight bytes needed to make each character are not stored consecutively. Looking at figure 2 you can see that the D-File is split in three sections of 64 lines each. Within each section, the eight lines which comprise each character are 256 bytes apart (8 lines * 32 character spaces). The junction of two sections is where there is a difference as the sequence begins to repeat. Study figure 1 to make this clear. I am told this unique structure has something to do with the way in which a TV draws its scan lines. Since I understand very little about the hardware, I must claim ignorance and accept the explanation.

This means that the easiest way to print to the screen is by using RST 1Øh, which is where Sinclair chose to start an all purpose print routine. Once again though, things are not as easy as they would seem. The 2Ø68 uses channels and streams to direct the traffic (we will discuss channels and streams later). This means that we must be sure we know where we are directing the output of RST 1Øh or else we will have no idea where it will end up.

Do not allow the D-File structure to put you off. We can still write to it if we understand its structure. Also, many of the routines we will need to help us handle it, are already located in the ROM.

Let's try a simple print using RST 1Øh. Enter listing 1 from the TS1ØØØ area above and run it with Rand Usr address.

You should get Error 5 on running this one. Notice how the bottom line is printed and scrolled. Probably not what you expected. We could call the channel open routine to fix this but, there is an upper/lower screen flag that can be temporarily set. If we reset bit Ø of TVFlag, we can print to the uppper screen. Insert as the first two instructions:

```
213C5C          Ld HL,TVFlag     ;get TVFlag address
36ØØ            Ld (HL),ØØ       ;reset flag
```

Now run the routine. Works great! A much better way is to only affect the bit needed. This requires the instruction Res Ø,(IY+Ø2), which we have not learned yet. You could also have achieved the same result with the first routine if you ran it with Print Usr address. This sometimes leads to undesireable results therefore, we will always use Rand Usr address or Let

x=Usr address.

Lets attempt to poke a character directly onto the screen. It cannot be done in one easy step as was the case with the TS1000. We must now resort to a complicated routine such as:

Listing 5

```
C630      MakeC  Add A,30h        ;offset to make number
                                  ;a printable character
ED4B365C  Print  Ld BC,(chars)    ;find character table
E5               Push HL          ;save character location
2600             Ld H,00h         ;transfer character
6F               Ld L,A           ;to HL
29               Add HL,HL        ;multiply by 8
29               Add HL,HL
29               Add HL,HL
09               Add HL,BC        ;get offset to character
                                  ;data in table
EB               EX DE,HL         ;address of data to DE
2AB05C           Ld HL,(Store)    ;we are storing address
                                  ;in D-File to print at
                                  ;in store
0608             Ld B,08h         ;# of lines/character
1A        Loop   Ld A,(DE)        ;get pixel data
77               Ld (HL),A        ;poke it to D-File
24               Inc H            ;adjust print pointer
13               Inc DE           ;adjust data pointer
10FA             DJNZ Loop        ;are we done? loop back
                                  ;if not, to complete
21B05C           Ld HL,Store      ;get and adjust print
34               Inc (HL)         ;position
E1               Pop HL           ;retrieve char location
C9        Done   Ret              ;one character printed
```

This routine is worthless without some data to print and another routine to setup the registers and call it. Notice that there are two entry points. Print is the normal entry however, MakeC is used to print a number without having its character code (as in raw data, instead of text). The unused location of 5CB0h in the system variables area stores the address of the next print position.

Upon entry to Print, we need to have the character to be printed in A. The HL register points to the character to print and needs to be preserved while Print is executing. Also, note that the program expects the data string to end with a byte containing FFh. Enter the following routine to set-up the registers and Call Print for a test. Your command to execute is Rand Usr Entry.

Listing 6

```
210040      Entry   Ld HL,4000h       ;1st address to print
22B05C              Ld (Store),HL     ;at into our variable
213075              Ld HL,Data        ;data string address
                                      ;assumed to be 7530h
7E          Loop    Ld A,(HL)         ;get character
FEFF                CP FFh            ;is it the end of
                                      ;string yet?
C8          Exit    Ret Z             ;ret if so
CD0080              Call Print        ;else go print it
                                      ;Print assumed to be
                                      ;at 8000h
23                  Inc HL            ;advance char pointer
18F6                Jr Loop           ;get next character
```

And here is the data as a hex dump:

Hex Dump 2

```
54 68 69 73 20 72 6F 75 74 69 6E 65 20 77 69 6C
6C 20 6F 6E 6C 79 20 70 72 69 6E 74 20 69 6E 20
6F 6E 65 20 74 68 69 72 64 20 6F 66 20 74 68 65
20 73 63 72 65 65 6E 20 61 74 20 61 6E 79 20 20
74 69 6D 65 2E FF
```

Be sure you have used the same addresses or change them to suit. If any address is not correct, you may crash.

Well, that's all folks. See ya next time.

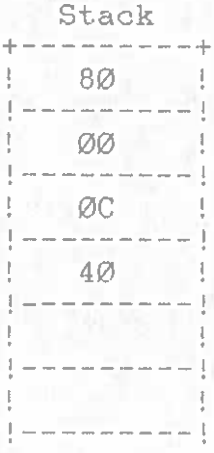
<END>

Syd Wyncoop
2107 SE 155th
Portland, Or 97233

Chart 1

Stack	! Exchange
Push rr	! Ex AF,AF'
Pop rr	! Exx
Ld SP,HL	! Ex DE,HL
Ld SP,nn	! Ex (SP),HL
Ld SP,(nn)	!
Inc SP	!
Dec SP	!

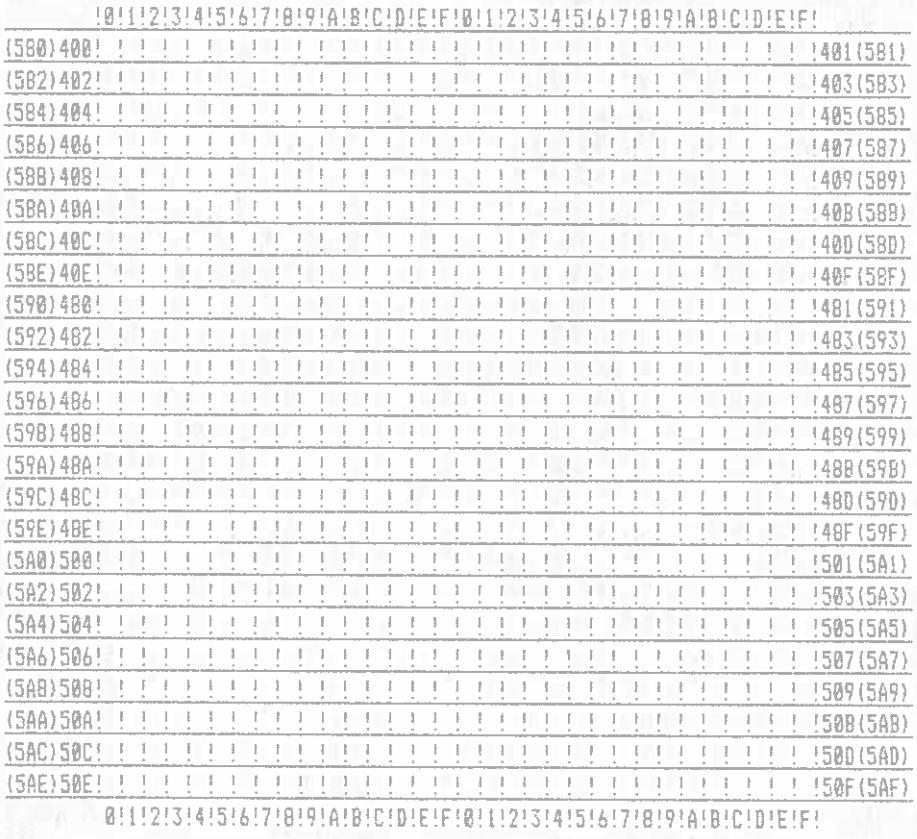
Figure 1



Assume the top two locations of the stack contain the address 8ØØØh, that the SP is set at 61FEh and HL contains ØC4Øh. A Push HL instruction will load ØC4Ø in the next two stack locations, 61FDh and 61FCh, and Dec SP twice, thereby making SP = 61FCh. A subsequent Pop HL instruction will then Inc SP twice while placing ØC4Øh into HL. It is important that even though SP has been adjusted, locations 61FCh and 61FDh still contain the address ØC4Øh and will until it is overwritten by another stack operation.

Figure 2

206B Display File Layout



Take first 3 digits from nearest edge
Take 4th digit from top/bottom edges
D-File addresses are not in brackets
A-File addresses are in the brackets

BEGINNING Z8Ø MACHINE CODE

LESSON 7

Before we begin, I need to ask again for some feedback from you. Especially if you are a TS1ØØØ user. I have heard from no TS1ØØØ users and will concentrate the programming on the 2Ø68 if you don't speak up! The MC instructions are the same for both computers however, each program must be taylored to the operating system. This makes writing this series more difficult. Also, I need your ideas. What would you like to see? We are near done with Z8Ø instructions.

Let's talk about the logical instructions, And, Or and Xor. And and Or do not give the true/false response you are familiar with if you have used them as Basic boolean operators. Instead, they and Xor operate on the individual bits of a register, or other 8 bit location. Also, the flags are always affected according to the result of these instructions.

Chart 1Ø provides the truth tables that explain how each of the logical instructions affects the bits being operated upon. While this makes the individual operations clear, it does little to help you understand the instruction, And FØh, when it is encountered. In order to understand these instructions better, it is neccessary to understand a little of Base 2 (binary) numbers.

As we discovered in our discussion of hex numbers, the highest digit in any base is equal to base-1. This means that we have 2 digits in binary, Ø and 1. The typical number 24Ø or FØh is 11110000b in binary. The b denotes a binary number just as our h means hex.

Rather than provide a full decimal to binary conversion chart, I have given you a hex to binary chart. This is because we have been working with hex numbers which are a very good shorthand for binary. Eight digit binary numbers are very easily represented by two digit hex numbers. I have provided program 1 for those of you wishing to generate your own charts.

I think you will agree that all those 1's and Ø's of binary are begging for us to make an error. That being true, why do we want to represent numbers in binary? The reason is because the logical instructions operate on individual bits and these bits can be easily represented as set or reset (on or off, if you prefer) which is 1 or Ø, respectively. Binary provides an easy way for us humans to determine how our friend CPU will act.

Let's look at And. And is often used to mask off unwanted bits. Suppose our routine puts the result in the accumulator and we want to insure that the result is never greater than 7. We would do this with the instruction, And Ø7h. If A contains 52h, the And Ø7h would make A contain Ø2h.

```
  A = Ø1Ø1ØØ1Ø = 52h
And   ØØØØØ111 = Ø7h
  A = ØØØØØØ1Ø = Ø2h
```

The result is always placed back in the A register. We have effectively said we only want to know about the three least significant bits of A, therefore we have discarded the rest.

Or is used to set the bits we need. If we wanted to insure the most significant bit is set we would Or 80h. If we wanted to insure the most significant bit is reset, we would And 7Fh. Can you see where 10000000b and 01111111b are more useful than 80h and 7Fh with these instructions? Binary allows you to see exactly what is happening.

```
A = 01010010 = 82h          A = 01010010 = 82h
Or  10000000 = 80h        And  01111111 = 7Fh
A = 11010010 = D2h          A = 01010010 = 82h
```

An example of using Or would be when we want to calculate an address. We would calculate the offset in A and then Or it with the high byte of the address to complete the calculation.

Xor is not a fugutive from the Outer-limits! It is a special case that sets only those bits that differ. For example:

```
A = 10010110 = 96h
Xor 01011101 = 5Dh
A = 11001011 = CBh
```

This is referred to as complimenting. Xor is complicated and is not used often but it is handy at times.

The bit manipulation instructions are the largest single group of Z80 instructions. They are Set, Res and Bit. They are easily understood as they set, reset or test the status of any bit in any register or address.

Set and Res are the set and reset instructions respectively and they do not affect any flags. These instructions are useful when you need to set/reset a bit without affecting the other bits in that byte. You could use And and Or to accomplish the same task but often you will not know the status of the other bits. Set and Res avoid this problem.

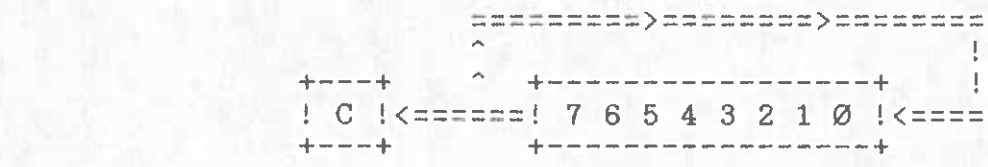
Bit is the test instruction. The bits are unchanged but the Zero flag is used to indicate the results of the test. The flag is set if the tested bit is zero, and reset if the bit is one.

The rotate and shift instructions are also bit manipulation instructions. They are classified separately as they operate on the entire byte. Many of them use the Carry flag to store a bit.

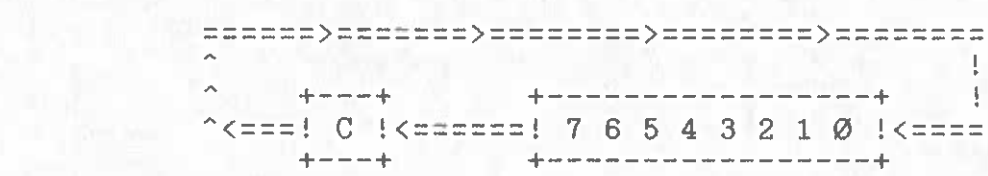
Rlca rotates the contents of the accumulator left one bit, placing the sign (most significant) bit in Carry as well as in bit 0. The effect of this instruction is to multiply A by 2. For example:

```
11001000b  becomes  10010001b
```

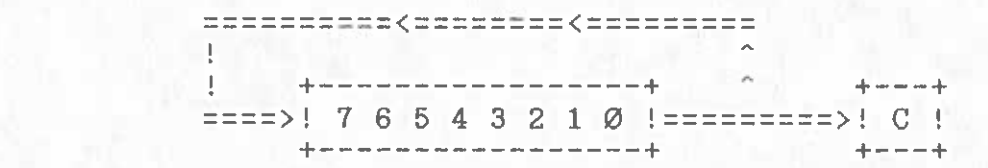
Graphically it looks like:



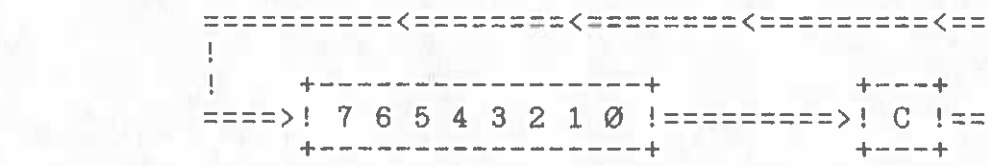
Rla rotates the accumulator bits left though Carry. The Carry flag still contains bit 7 but the prior Carry flag is now in bit Ø. Here it is graphically:



Rrca is similiar to Rlca. This time the accumulator's bits are rotated right. Bit Ø is copied into Carry and bit 7. The effect of this instruction is to divide A by 2. This one looks like:

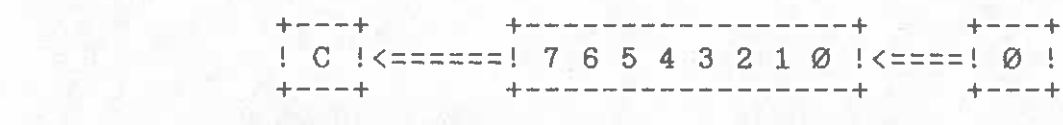


Rra is not surprisingly like Rla, except that we are rotating right. Bit Ø is rotated through the Carry flag. Here it is:



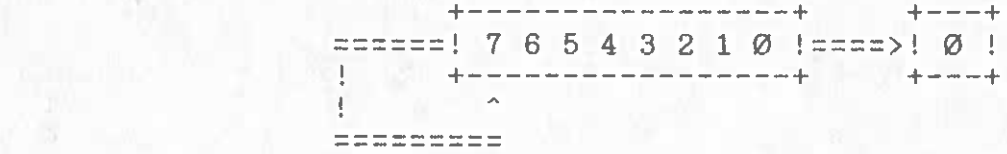
The remaining rotate instructions will operate on any register (including A) or the contents of any address. Rlc r is graphically the same as Rlca. Rl r is graphically the same as Rla. Rrc r is graphically the same as Rrca. Rr r is graphically the same as Rra. The difference between Rra and Rr A is that Rra affects only the Carry flag while Rr A affects all the flags.

The shift instructions are the true arithmetic instructions although they are otherwise similiar to the rotate instructions. The first, Sla is similiar to Rlc instruction, except that the least significant bit becomes zero. The effect is to multiply the register or address contents by two. Graphically we have:



Sra will shift right arithmetic the bits in the specified register or address. This is similiar to Rrc except that bit Ø is only copied to the Carry flag. Bit 7 remains as it was. The effect of this is to divide signed numbers by two, leaving the carry set if there was a remainder (you were dividing an odd

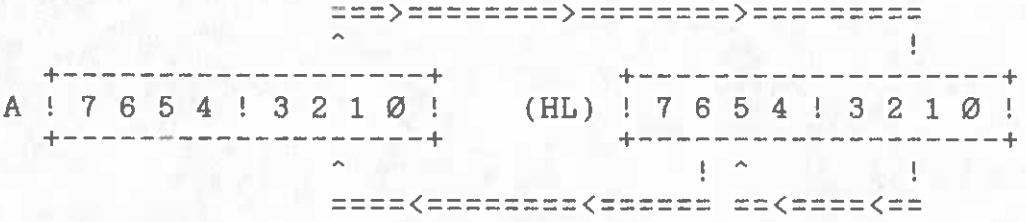
number). Graphically, we have:



Srl, or shift right logical is the same as Sra, except that the most significant bit becomes zero. Graphically, this is the reverse of Sla:

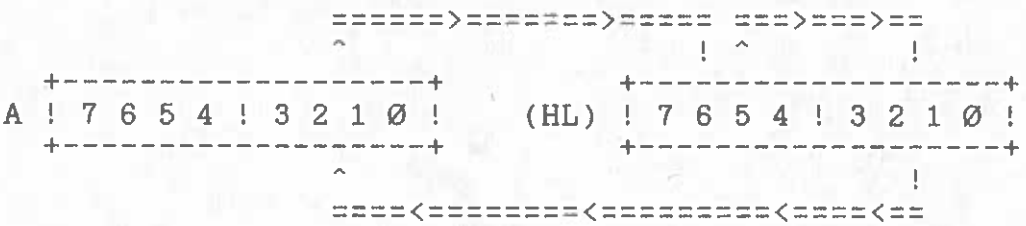
The last two shift instructions, I have never found a use for. They are Rld and Rrd, which mean rotate left decimal and rotate right decimal, respectively. They operate on the contents of the memory location addressed by HL and the accumulator.

In the case of Rld (not to be confused with Rl d) the low nybble of (HL) is copied into the high nybble of (HL), the high nybble of (HL) is copied into the low nybble of the accumulator, and the low nybble of the accumulator is copied into the low nybble of (HL). Got it? Here's a picture:



For example, assume A contains 7Ah and (HL) contains 31h. After an Rld instruction, A will contain 73h and (HL) will contain 1A.

Rrd behaves just as obnoxiously except, of course, that the rotation is to the right. Here it is:



Remember, you will be limited to an eight bit answer with these instructions. The Carry flag will indicate an overflow and the accumulator, register or memory location will contain the difference. In other words, all arithmetic results will be modulo 256.

Now, how about a practical application? Let's develope a hex dump routine. We can show any byte as two hex digits once we know where to begin. We need the Basic interface first:

Hex Dump Interface

```
  1Ø Print "Dump from decimal
address: ";
  2Ø Input a
  3Ø Print a
  4Ø Print
  5Ø Poke base-1,Int (a/256)
  6Ø Poke base-2,a-Int
(a/256)*256
  7Ø Rand Usr base
  75 Rem base=address of Hex
Dump,substitute your addresses
for base
  8Ø If Inkey$="" Then Goto 8Ø
  9Ø If Code Inkey$=13 Then Goto
7Ø
  95 Rem 13=Enter on the TS2Ø68
     Use 118 on the TS1ØØØ
 1ØØ If Inkey$="Z" Then Copy
 11Ø Goto 8Ø
```

And now the Hex Dump routine for the TS2Ø68. Remember to use your addresses in place of the xxxx.

```
          Store   Equ HexDmp-2
FDCBØ286  HexDmp  Res Ø,(TVFlag)   ;this bit tells the Rom
                                   ;routine to print in the
                                   ;main screen area
2Axxxx            Ld HL,(Store)    ;get address to begin dump
ØE1Ø              Ld C,1Øh         ;counter-# of lines to dump
7C        OutrLp  Ld A,H           ;get the high byte of the
                                   ;first address in this line
                                   ;of the dumped bytes
CDxxxx            Call HexPrt      ;go print it
7D                Ld A,L           ;get low byte of address
CDxxxx            Call HexPrt      ;go print it
3E2Ø              Ld A,2Øh         ;ascii space
F5                Push AF          ;save the space character
D7                Rst 1Øh          ;print the space
F1                Pop AF           ;retrieve the character
D7                Rst 1Øh          ;print another space
Ø6Ø8              Ld B,Ø8h         ;counter-# bytes/line
7E        InnrLp  Ld A,(HL)        ;byte to accumulator
CDxxxx            Call HexPrt      ;go print it
3E2Ø              Ld A,2Øh         ;ascii space
D7                Rst 1Øh          ;print the space
23                Inc HL           ;advance byte pointer
1ØF6              Djnz,InnrLp      ;loop for 8 bytes
3EØD              Ld A,ØDh         ;ascii carriage return
D7                Rst 1Øh          ;go to start of next line
ØD                Dec C            ;count one line done and
2ØEØ              Jr nz,OutrLp     ;loop for 16 lines
22xxxx            Ld (Store),HL    ;store start of next line
```

```
C9          Done    Ret                 ;return to basic

F5          HexPrt  Push AF             ;save it
E6FØ                And FØh             ;mask off high nybble
1F                  Rra                 ;and rotate to low nybble
1F                  Rra
1F                  Rra
1F                  Rra
CDxxxx              Call Print          ;go print digit in A
F1                  Pop AF              ;retrieve it
E6ØF                And ØFh             ;mask off low nybble
CDxxxx              Call Print          ;go print digit in A
C9                  Ret                 ;ret to calling routine

FEØA        Print   Cp ØAh              ;check if digit is greater
                                        ;than 9
3F                  Ccf                 ;if so, set carry and
DCxxxx              Call c,Offset       ;go adjust for correct
                                        ;ascii character codes
C63Ø                Add A,3Øh           ;make a printable char code
E5                  Push HL             ;save registers
C5                  Push BC
D7                  Rst 1Øh             ;Rom print routine
C1                  Pop BC              ;restore registers
E1                  Pop HL
C9                  Ret                 ;return to calling routine

C6Ø7        Offset  Add A,Ø7h           ;adjust digit to skip over
                                        ;ascii characters between
                                        ;9 and A
C9                  Ret                 ;return to calling routine
```

And for you TS1ØØØ owners (I still use mine!):

```
2A7B4Ø      HexDmp  Ld HL,(Store)       ;get address to begin dump
ØE1Ø                Ld C,1Øh            ;counter-#of lines to dump
7C          OutrLp  Ld A,H              ;get the high byte of the
                                        ;first address in this line
                                        ;of the dumped bytes
CDA94Ø              Call HexPrt         ;go print it
7D                  Ld A,L              ;get low byte of address
CDA94Ø              Call HexPrt         ;go print it
AF                  Xor A               ;same as Ld A,ØØh
F5                  Push AF             ;save the space character
D7                  Rst 1Øh             ;print the space
F1                  Pop AF              ;retrieve the character
D7                  Rst 1Øh             ;print another space
Ø6Ø8                Ld B,Ø8h            ;counter-# bytes/line
7E          InnrLp  Ld A,(HL)           ;byte to accumulator
CDA94Ø              Call HexPrt         ;go print it
AF                  Xor A               ;get space character
D7                  Rst 1Øh             ;print the space
23                  Inc HL              ;advance byte pointer
1ØF7                Djnz,InnrLp         ;loop for 8 bytes
3E76                Ld A,76h            ;get carriage return char
D7                  Rst 1Øh             ;go to start of next line
```

```
ØD                  Dec C               ;count one line done and
2ØE2                Jr nz,OutrLp        ;loop for 16 lines
227B4Ø              Ld (Store),HL       ;store start of next line
C9          Done    Ret                 ;return to basic

F5          HexPrt  Push AF             ;save byte
E6FØ                And FØh             ;mask off high nybble and
1F                  Rra                 ;rotate to low nybble
1F                  Rra
1F                  Rra
1F                  Rra
CDBA4Ø              Call Print          ;go print digit in A
F1                  Pop AF              ;retrieve byte
E6ØF                And ØFh             ;mask low nybble
CDBA4Ø              Call Print          ;go print it
C9                  Ret                 ;return to calling routine

C61C        Print   Add A,1Ch           ;make a character Ø to F
E5                  Push HL             ;save registers
C5                  Push BC
D7                  Rst 1Øh             ;Rom print routine
C1                  Pop BC              ;restore registers
E1                  Pop HL
C9                  Ret                 ;return to calling routine
```

I have to assume that if you are still with me, you have obtained some good study aids. Since almost all books on the subject of Z8Ø MC have numerous tables in them, this is the last time I will give the hex codes in the MC disassemblies. I will instead, provide the source code. What's source code? That's another lesson. See ya soon!

<END>

Syd Wyncoop
21Ø7 SE 155th
Portland, Ore 97233

Truth Tables

And			Or			Xor		
	Ø	1		Ø	1		Ø	1
Ø	Ø	Ø	Ø	Ø	1	Ø	Ø	1
1	Ø	1	1	1	1	1	1	Ø

Hex/Bin Conversions

Hex	Bin	Hex	Bin
Ø	ØØØØ	8	1ØØØ
1	ØØØ1	9	1ØØ1
2	ØØ1Ø	A	1Ø1Ø
3	ØØ11	B	1Ø11
4	Ø1ØØ	C	11ØØ
5	Ø1Ø1	D	11Ø1
6	Ø11Ø	E	111Ø
7	Ø111	F	1111

Chart 11

Logical	Rotate and Shift
And r	Rlca
And n	Rla
And (HL)	Rrca
Or r	Rra
Or n	Rlc r
Or (HL)	Rlc (HL)
Xor r	Rl r
Xor n	Rl (HL)
Xor (HL)	Rrc r
	Rrc (HL)
	Rr r
Bit Manipulation	Rr (HL)
Bit b,r	Sla r
Bit b,(HL)	Sla (HL)
Set b,r	Sra r
Set b,(HL)	Sra (HL)
Res b,r	Srl r
Res b,(HL)	Srl (HL)
	Rld
	Rrd

Program 1

```
 1Ø LPrint "Dec  Hex  Bin"
 2Ø For i=Ø to 255
1ØØ Let h$="  "
11Ø Let h1=Int (i/16)
12Ø Let h$(1)=Chr$ (h1+48+(7
And h1>9))
13Ø Let h2=i-h1*256
14Ø Let h$(2)=Chr$ (h2+48+(7
```

```
And h2>9))
 200 Let b$="ØØØØØØØØ"
 210 Let a=i
 220 For n=7 to Ø Step -1
 230 If (a-2*Int (a/2)) Then Let
b$(n+1)="1"
 240 Let a=Int (a/2)
 250 Next n
 300 Let t=(1 And i<10)+(1 And
i<100)+(Ø And i<1000)
 310 LPrint Tab t;i;Tab 5;h$;Tab
10;b$
 320 Next i
```

TS1000 users need to change the following lines:

```
 120 Let h$(1)=Chr$(h1+28)
 140 Let h$(2)=Chr$(h2+28)
```

BEGINNING Z8Ø MACHINE CODE

LESSON 9

It has been pointed out to me, by an astute reader, that I neglected to tell you to run your MC routines in SLOW, if you are using the TS1ØØØ. Otherwise, you cannot see the display of any of my examples. Sorry about that.

This time we will discuss the I/O instructions. For those of you that are wondering what I/O means, it is Input and Output. When I was new to computerdom, I thought I/O referred to my financial status.

To what are we Inputing and Outputing? The computer, but it is actually our old friend, CPU. The I/O instructions allow the CPU to receive or send information through the concept of PORTS and they accomplish this depending upon how the manufacturer made the hardware surrounding the CPU. For example, in our computers port FEh is used for the keyboard. There are others used by Sinclair for the 2Ø4Ø printer, cassette, and on the 2Ø68, for the bank switching and video mode changes. These are 'hard-wired' in the computer and supported by the operating system, therefore we cannot change them.

What is a PORT? Very simply, it is the doorway though which information flows to and from the CPU and outside devices. There are 256 ports available to us on the Z8Ø (there are really 65,536, but we will not consider them here). The ports are of course numbered Ø-255, as they must be referred to in a single byte. Think of each port as a door to a storeroom. Each door has a number on it, much like a motel would. Each storeroom can hold one byte of data at a time. The CPU can put data in or take data out, by referring to each port (door).

The I/O instructions are In and Out, respectively and there are two forms of the instructions, as detailed in the syntax chart. We are looking at some instructions that are almost english and fairly easily understood.

The forms In A,(n) and Out (n),A use the port specified by n and reads (In) data into A or writes (Out) data from A. This is very similar to the Basic In and Out commands, except that the data is stored in the accumulator. None of the flags are affected by these instructions.

For example:

In A,(FFh)	reads port FFh and places one byte of data into the accumulator
Out A,(FFh)	writes one byte of data from the accumulator to the device which is addressed by port FFh

The forms In r,(C) and Out (C),r allow the flexibility of reading or writing data with any register. Caution, remember that C contains the port address. The Out (C),r instruction does not affect any flags, while the In r,(C) affects all the flags,

except Carry, according to data which was read in.

Register C must be loaded with the port address, prior to use, as in these examples:

```
Ld C,FFh        reads port FFh and places data
In B,(C)        in the B register

Ld C,FFh        writes data from the B register
Out B,(C)       to the device addressed by port FFh
```

You will note the I/O instructions assume you are communicating with some device (printer, monitor, disk, etc.) which is 'addressed' by a port number. The port number is selected by the hardware manufacturer, just as Sinclair did in our computers. You can perform I/O operations on all ports however, the results are unpredictable without a device attached. This is due to lack of pull-up resistors on the data lines. Obviously, there will not be any communication if there is no device attached or an incorrect port number is used.

Since we have to contend with devices that are much slower than the CPU, we also have to consider timing. I will not get into this subject very deep, as this type of programming becomes very hardware dependent.

The timing problem is obviously one of slowing down the I/O operations, in an effort to match the device. Let's consider the simple case of reading a switch. We might wish to read the switch once per second, to eliminate multiple switch closures (a good example is in debouncing the keyboard switches).

We can perform this type of delay by looping for a predetermined time period. A simple delay routine that can be used, without destroying any registers is:

```
Delay   Push BC         ;save these registers
        Ld B,xx         ;xx = # of ms to delay
Dly1    Ld C,yy         ;yy = 1 ms delay count
Dly2    Dec C           ;loop for 1 ms
        Jr NZ, Dly2
        DJNZ, Dly1      ;loop for # of ms
        Pop BC          ;retrieve registers
        Ret             ;end delay
```

The value xx is the number of milliseconds to delay and yy is the number of loops needed for a delay of one millisecond. I will not take you through the steps of counting the delay as I wish only to demonstrate the technique. When you are ready to use this routine, you will not need my help with the values xx and yy.

Another method of delay can be used with 'smart' devices, such as a printer. This method uses two separate loops, instead of the nested loops we just looked at. Our example assumes the printer (actually, it's interface) is wired for port 7Fh and it sends a zero byte when ready to accept data.

```
Ready?  Ld C,7Fh        ;get port address
        In A,(C)        ;get ready status from printer
        Jr NZ,Ready?    ;loop unless zero byte received
Print   Ld A,data       ;get data byte to print
        Out (C),A       ;send data to printer
```

```
Ret             ;end delay
```

This method has the advantage of not sending any data, unless the device is ready, therefore no data is lost. Can you imagine how this article would look, if some characters were lost in transit to the printer? No, that's not what happened, I just write poorly.

There is another solution to this timing problem, which uses hardware. We will not discuss that here, but you should be aware of it.

You also note some I/O instructions on the chart that I have not explained. These perform block I/O operations and will be explained next time, with the rest of the block instructions. They are included here so that it will be clear they are I/O instructions.

By now, many of you are undoubtedly trying to write your own MC programs. I wish to give some tips and hints, that will make the process less painful.

First, DO NOT attempt to write a large MC program on the first try. Instead, take the approach we have followed here and write small routines that do a specific job. They can be easily called from Basic and will return to the next Basic line. I would suggest you take a small working Basic subroutine and try converting it to MC. An arithmetic routine is the easiest to convert, as long as it does not contain special functions, such as SQR, COS, etc.

Write your MC in modules (subroutines) that can be easily tested and debugged. This also allows you to develope a library of known, debugged routines that can be used again. Look closely at the routines I have provided in this series. You will note that they are very similiar to each other.

I do not flow-chart and will not describe that to you. There are many good books on the subject. However, before you begin codeing your routine, there are some questions you need to answer or data to collect:

1)Purpose -what do we hope to accomplish?
2)Examples -what happens if? try several tests
3)Inputs -what data does routine need upon entry?
4)Outputs -what data is returned to calling program?

I also strongly encourage you to document your program. All of us have purchased programs that were not user friendly, in spite of it's claims, and in addition, had no documentation. This is deplorable, but the biggest reason for documenting your own programs is for ease of use. I have written code, been interrupted, and when I returned to it a few months later, I could not determine what the code did or why I wrote it that way! Some essentials to proper documentation include:

1)Purpose -if the above questions were answered, they should be included here
2)Registers -which ones are used? which ones are destroyed and which are preserved? what should they contain upon entry and exit?
3)Inputs -what data does routine need upon entry?
4)Outputs -what data is returned to calling routine?
5)Routines Callled By -what routines use this one as a subroutine?
6)Routines Called -what routines does this one call?

7)Commented Source -an absolute necessity

This is not the only information that should be in your documentation, but it is enough to make that routine useful to you next time around. Without this information, you will not develope a useful library of routines and will continually need to reinvent the wheel. If you follow these suggestions, you will find MC programming easy (well, almost) and if not, you will soon give up in frustration.

Now, how about a short routine? Let's convert binary numbers to decimal digits for printing.

The easiest way to accomplish this is by repetitively subtracting powers of ten from our binary number and counting the number of times the subtraction is possible, better known as division. For the more advanced, try doing this by using the shift and rotate instructions. I am using the subtraction technique, as the code is much easier to follow.

```
;Set-up Demonstration
;********************
;
;Inputs: none
;Outputs: print decimal number
;Routines Called: Bn2Dec
;Routines Called By: none
;Purpose: set-up hl for our conversion routine
;
        Org 7530h
Set-up  Ld HL,4000h   ;hl=number to convert
        Call Bn2Dec   ;go convert it
Done    Ret           ;converted and printed
                      ;this is our return to
                      ;basic
;
;
;our routine really begins here
;
;Convert Binary to Decimal
;*************************
;
;Inputs: HL=Binary Number
;Outputs: decimal number is printed
;Routines Called: Divide
;                 Print
;Routines Called By: Set-up
;Purpose: convert binary number to decimal
;         ascii characters for printing
;
Bn2Dec  Ld BC,D8F0h   ;-10,000
        Call Divide   ;go get 10^4 digit
        Ld BC,FC18h   ;-1,000
        Call Divide   ;go get 10^3 digit
        Ld BC,FF9Ch   ;-100
        Call Divide   ;go get 10^2 digit
        Ld BC,FFF6h   ;-10
        Call Divide   ;go get 10^1 digit
        Ld A,L        ;a = 10^0 digit
Exit    Jp Print      ;go print 10^0 digit
;
```

```
;
;Divide by 10^x
;**************
;
;Inputs: HL=Binary Number
;        BC=10^x
;Outputs: A=decimal digit to print
;Routines Called: Print
;Routines Called by: Bn2Dec
;Purpose: divide binary number by power
;         of ten to obtain decimal digit
;         by repetitive subtraction
;
Divide  Xor A          ;clear our counter
DvLoop  Add HL,BC      ;perform subtraction
        Inc A          ;count it
        Jr C,DvLoop    ;do again if possible
        Sbc HL,BC      ;otherwise adjust the
        Dec A          ;counters for the extra
                       ;subtraction
        Ret Z          ;division not possible
DvDone  Jp Print       ;go print it
;
;
;Print Ascii Character
;********************
;
;Inputs: A=decimal digit
;Outputs: digit in A is printed
;Routines Called: Rom Print
;Routines Called By: Bn2Dec
;                    Divide
;Purpose: call rom print routine while
;         preserving the registers
;
Print   Push HL        ;save all registers
        Push BC
        Add A,30h      ;2068 only
                       ;make an ascii character
        Add A,1Ch      ;1000 only
                       ;make an ascii character
        Rst 10h        ;go print digit in A
        Pop BC         ;restore registers
        Pop HL
PrDone  Ret            ;digit has been printed
;
End
```

There are several things to note. First, since each routine is a separate module that could be called from anywhere, there are a few unneccessary instructions. For instance, the Jp Print is not needed at the DvDone label, as the Divide routine could simply 'fall through' to Print. I used Jp to demonstrate the use of another routine's Ret instruction, in place of a Call Print and the subsequent Ret that would have been needed to end the Divide routine. Assume for a moment that Print does not follow

immediately behind Divide. Try to follow the program through and see that the Divide routine uses the Ret instruction from the Print routine to return to the main routine, Bn2Dec.

Also, I used the Rst 10h rom print routine for compatibility on both the 1000 and 2068. Use of the rom routines often destroys the registers, therefore they were saved. Even BC, which we could have discarded.

The source is written along the quidelines given above. You should note that the comments do not echo the instructions, except when it serves to clarify. I have seen many listings that look like:

```
        Ld HL,4000h     ;HL=4000h
```

Obviously not very informative or useful.

Several lessons ago, I made the rather obnoxious claim that all arithmetic could be performed with addition. This routine will perhaps clarify that statement. We needed to divide. We chose to subtract, to achieve this. We chose to add a negative number, in lieu of subtraction. We divided!

As a friend of mine says, "Th-Th-Th-That's all folks!", that is, until next issue.

<END>

Chart 1

In A,(n)	Out A,(n)
In r,(C)	Out (C),r
Ini	Outi
Inir	Otir
Ind	Outd
Indr	Otdr

Syd Wyncoop
2107 SE 155th
Portland, Or 97233

BEGINNING Z8Ø MACHINE CODE

LESSON 1Ø

The subject this time is the Z-8Ø Block instructions. There are block instructions for I/O, search (compare), and transfer (assignment). We listed the block I/O instructions last lesson but they are detailed again in chart 1.

Before we look at the instructions, we need to review one of the Z-8Ø's flags. It is the parity/overflow (P/V) flag and is an overworked little devil, as it keeps track of two conditions, depending upon the instruction being executed. I gave you a chart of affected flags, by instruction, in lesson 5 (if you need lesson 5, contact TDM for a back issue!)

Overflow is similiar to carry except that it occurs only when there is a carry from bit 6 to bit 7, of the accumulator, in signed arithmetic. The effect of an overflow is to change the sign bit of the accumulator. Overflow can be detected by use of the carry flag, but it is more difficult.

The use of the P/V flag we are interested in is Parity. Parity is either even or odd and is simply a count of the set bits in a byte or register. An even number of set bits results in even parity and a set parity flag. Parity is indicated with the logical, rotate, I/O and all block instructions.

The actual use of the parity flag in the block instructions is to indicate when the BC register pair has been decremented to Ø (see below). You will recall that 16 bit decrements do not affect the zero flag. Since the Z-8Ø can indicate BC=Ø in the P/V flag, it could have done the same in the zero flag, except that the zero flag already has a use in the block instructions (see below).

There is one last piece of information we need in order to use the block instructions; how and which registers do we need to set-up? All the register pairs are used as follows.

The BC pair is a 16 bit counter. The parity flag is set and the block instruction is terminated when BC=Ø. There is no 8 bit counter allowed, except for the I/O instructions, where B serves the purpose.

The DE pair is a DEstination pointer for block memory transfers.

The HL pair, as usual, is a memory pointer for all the block instructions.

All the block instructions decrement BC and either increment or decrement DE and HL, according to the type of instruction. The third letter of the mnemonic will be 'i' for and increment or 'd' for decrement.

If the fourth letter of the mnemonic is an 'r', then the instruction is functionally the same as the the 3 letter version, except that the instruction repeats until a counter has been decremented to Ø.

Now for the instructions. I have listed the instruction (a few samples for each group) with its operation broken into 'equivalent' instructions, next to it. REMEMBER, the equivalent instructions are for clarification ONLY and are not executable!

The first set is the completion of our I/O instructions, from last lesson.

```
        Ini                    Indr
---------------        ---------------
   Ld (HL),(C)        Loop Ld (HL),(C)
   Inc HL                    Dec HL
   Dec B                     DJNZ Loop
```

Notice that the block instruction is the same as the In r,(C) instruction from last lesson. The difference is that r can only be (HL) and the B register is a counter, hence the above 'equivalent' instructions.

Note also, how the auto repeat works. Since the repeat is part of the instruction, no other operation can occur in the loop (except, of course, interrupts--but that's next lesson). The loop is not exited until B=Ø.

The block Out instructions are the same except that the byte pointed to by HL is moved Out port (C).

The block search instructions are a variation of our old friend Cp (compare), as follows:

```
        Cpd                    Cpir
---------------        ---------------
   Cp (HL)            Loop Cp (HL)
   Ret Z                     Ret Z
   Dec HL                    Inc HL
   Dec BC                    Dec BC
                             Jr NZ,Loop
```

Note the additional exit point (Ret Z). These are called the block search instructions, as they will look at each byte and set one of two flags. The zero flag is set if A=(HL), (there is no Ret to anything) or the parity flag is set if BC=Ø. Since the Ret Z is for demonstration only, it is important to know that the operations on BC and HL will occur, even if a match has occurred. Therefore, you may need to adjust a pointer, after a match.

For example, assume the accumulator contains FFh, HL contains 4ØØØh and BC=Ø6h. This is the section of memory to search:

Address	Contents
4ØØØh	ØØh
4ØØ1h	Ø9h
4ØØ2h	F9h
4ØØ3h	FFh
4ØØ4h	C9h
4ØØ5h	E1h

The search will end with the match at address 4ØØ3h and the registers will contain:

```
A  = FFh
HL = 4ØØ4h
BC = Ø1h
```

The zero flag will be set, to indicate a match, and the

parity flag will not be set, as we did not reach zero in BC.

The last group of block instructions are for memory transfers (move one block of memory from here to there). They are essentially a variation on the assignment instructions (Ld) except that they work on two memory locations, instead of a register and a memory location.

The registers must be set-up in advance for these instructions to work properly, as follows:

```
BC = size of block to transfer
HL = first byte address of block to transfer
DE = first byte address of new location of
     block, after transfer (DEstination)
```

Once the registers are set-up, the instructions work like this:

```
        Ldd                     Ldir
----------------        ----------------
   Ld (DE),(HL)        Loop Ld (DE),(HL)
   Dec DE                   Inc DE
   Dec HL                   Inc HL
   Dec BC                   Dec BC
                            Jr NZ,Loop
```

Note that we have only one exit to the loop, the case where BC=Ø.

The following routine should be placed in your Ø REM statement, to move your MC above Ramtop:

```
Move Ld HL,Base       ;start address of your MC
     Ld DE,Ramtop+1   ;destination address above
                      ;Ramtop, where your MC will run
     Ld BC,Length     ;length of your MC routine
     Ldir             ;move your MC above Ramtop
     Ret              ;back to Basic
```

One important point, any absolute addresses (Call nnnn, Jp nnnn, etc.) must be adjusted to indicate locations within the new block. The usual method is to assemble your MC to run at its correct location, then place it in the Rem statement for storage and SAVEing. This is the better method of saving and running MC from high memory on the TS1ØØØ, than the method I gave last lesson. See if you can make a small change in the above routine to move your MC from high memory to your Ø REM statement, using the Lddr instruction.

The last caveat to watch for with transfers is overwriting a portion of your MC, if the blocks overlap. When there is an overlap of blocks, the bytes can often only be moved in one direction or from one end of a block. For example, the routine above moves a block from start to end. It could just as easily been moved from end to start, using the Lddr instruction, if the pointers indicated the end of each block.

The astute reader will begin to see some possibilities in these instructions, as they are fast and very powerful. You could, for instance, write your own 'find and replace' routines, create 'instant' screen swaps or even animate a small section of the display (sprites). I'll leave you with your imagination and the following routine.

Our routine deviates from the instructions of this lesson. It is a renumbering routine for Basic programs and is given as a demonstration of what is possible and give you some more technique. It will renumber any Basic program from a stated line (which must exist) to the program end.

Many of the routines can be used in other programs, such as the input routine. It uses some error checking in order to avoid any non-numeric input. It also gives the method of converting an Ascii string of digits to a binary number for use in calculations. It does however lack a backspace or delete. Can you see how to add it by reading one additional key press and adjusting the buffer pointer? Notice how the carry flag is used to indicate an error. Also, note that space must be left at the end of the routine for the input buffer. Do you want prompts anywhere on the screen? Run the Input routine with a PRINT USR address!

Note that this program is written in rather large modules, that fall through to the next one. It is extremely hard to debug a program written in this fashion, unless you are using routines that are known to be bug-free. Can you see the obvious places for break-points, in order to test for debug purposes?

Note how we reuse the string data for the Renumber prompt. But, enough of this. Here's the routine:

```
;***********************************************
;              RENUMBER BASIC PROGRAM
;***********************************************
;
;Basic system variables:
Prog    Equ 5C53h
LastK   Equ 5C08h
;
;ROM calls:
KeyScan Equ 02E1h       ;TS1000 = 02BBh
LneAddr Equ 16D6h       ;TS1000 = 09D8h
DeCode  Equ 07BDh       ;TS1000 only
;
Org     FC00h           ;TS1000 = 7C00h
;
;Test for a Basic Program to renumber for the TS2068
;
Start   LD HL,(Prog)    ;no program line number has the
        BIT 7,(HL)      ;7th bit set in high byte of line
                        ;number, but start of VARS does
        RET NZ          ;no program-return to Basic
;
;Test for a Basic Program to renumber for the TS1000
;
Start   LD HL,407Dh     ;start of Basic program area will
        LD A,76h        ;contain an ENTER (chr$ 118) if no
        CP (HL)         ;program as will be first character
        RET Z           ;of the display file
;
;This is common code for the 1000 and 2068
;
;Get data for renumbering
;
G_From CALL PrRnum      ;prompt for Renumber from line #
        LD HL,FromLn
        CALL Print
        CALL Input      ;go get line #
        JR C,G_From     ;bad input-do it again
        LD HL,OldLine   ;save input in this variable
        LD (HL),E
        INC HL
        LD (HL),D
;
;Now, get the first new line number
;
G_New   LD HL,NewLn     ;prompt for Start with new line #
        CALL Print
        CALL Input      ;go get line #
        JR C,G_New      ;bad input-do it again
        LD HL,NewLine   ;save input in this variable
        LD (HL),E
        INC HL
        LD (HL),D
;
;And, finally the step for the new line numbers
;
G_Step CALL PrRnum      ;prompt for Renumber in steps of
        LD HL,Incr
```

```
        CALL Print
        CALL Input        ;go get step in lines
        JR C,G_Step       ;bad input-do it again
        LD HL,Step        ;save input in this variable
        LD (HL),E
        INC HL
        LD (HL),D
;
;Search for first line to renumber
;
Search LD HL,(OldLine) ;set-up HL for Rom routine that
        CALL LneAddr      ;returns the address of the line
                          ;whose number is held in HL, in the
                          ;HL register pair, or the line that
                          ;follows it, if it does not exist.
                          ;The start of the previous line is
                          ;returned in DE. The zero flag is
                          ;set if the line number was found.
        JR Z,ReNumb       ;found it-ok to continue
        LD HL,NotFnd      ;not found-give error msg
        CALL Print
        RET               ;and return to Basic
;
;Begin renumbering
;
ReNumb LD DE,(NewLine) ;get the next new line #
        LD (HL),D         ;load it into the present
        INC HL            ;line # bytes
        LD (HL),E
        INC HL            ;advance pointer
        PUSH HL           ;save it
        LD HL,(Step)      ;get step between line #'s
        ADD HL,DE         ;and adjust the next line #
        LD (NewLine),HL ;put next line # back in variable
        POP HL            ;retrieve pointer
        LD E,(HL)         ;get line length into DE
        INC HL
        LD D,(HL)
        INC HL            ;adjust pointer to start of Basic
                          ;line (after line # and length)
        ADD HL,DE         ;add line length to pointer to
                          ;adjust for start of next line
;
        BIT 7,(HL)        ;test for valid line # or  ;2068 only
                          ;start of Basic variables  ;2068 only
        RET NZ            ;return to Basic, if done  ;2068 only
;
        LD A,76h          ;test for valid line # or  ;1000 only
        CP (HL)           ;start of D-File           ;1000 only
        RET Z             ;return to Basic, if done  ;1000 only
;
        JR ReNumb         ;go do next line
;
;Print routines
;
PrRnum LD HL,Renum     ;special entry to print the word
                          ;Renumber (this saves data space)
Print  LD A,(HL)          ;HL=pointer to step thru messages
```

```
        CP FFh          ;check for terminating byte and
        RET Z           ;exit routine if found
        PUSH HL         ;save pointer
        RST 10h         ;rom print routine
        POP HL          ;retrieve pointer
        INC HL          ;and adjust it
        JR Print        ;loop to print next character
;
;Input routine
;
Input   LD HL,Buffer    ;storage for input
        LD (Pointr),HL  ;reset buffer pointer-effectively
                        ;clearing the buffer
;
;This is for the TS2068 only
;
ScanKy  LD A,FFh        ;clear last input character
        LD (LastK),A
        CALL KeyScan    ;use rom routine to get key pressed
        LD A,(LastK)    ;get newly pressed key code
;
;This is for the TS1000 only
;
ScanKy  CALL KeyScan    ;use rom routine to get key pressed
        INC L           ;and check for heavy-handed human
        JR NZ,ScanKy    ;to lift finger
NewKey  CALL KeyScan    ;use rom routine to get key pressed
        PUSH HL         ;which is returned in HL but,
        POP BC          ;is needed in BC for DeCodeing
        INC L           ;check and wait for a new key press
        JR Z,NewKey
        CALL DeCode     ;rom routine to decode key press, HL
                        ;will point to proper key in the rom
                        ;key table
        LD A,(HL)       ;put keycode into A from table
;
;This is common code for the 1000 and 2068
;
        CP 0Dh          ;accept ENTER (1000=76h)
        JR Z,EndInp     ;and end input if so, else
        CP 30h          ;check for and accept only (1000=1Ch)
        JR C,ScanKy     ;the digits 0 to 9, else
        CP 3Ah          ;(1000=26h)
        CCF
        JR C,ScanKy     ;continue scanning the keyboard
DigtOk  LD HL,(Pointr)  ;input has been accepted-retrieve
        LD (HL),A       ;buffer pointer and store digit
        INC HL          ;adjust pointer for next digit
        LD (Pointr),HL  ;and save it
        RST 10h         ;echo accepted key press to screen
        JR ScanKy       ;continue input
EndInp  LD HL,(Pointr)  ;retrieve the buffer pointer
        LD (HL),A       ;store ENTER in buffer
        RST 10h         ;advance print position to next line
                        ;on the screen
                        ;
                        ;we now have accepted, verified and
                        ;ended our input but it needs to be
```

```
                        ;converted from a string of Ascii
                        ;characters to a single word Binary
                        ;number.
                        ;
Asc2Bi  LD HL,Buffer    ;get start of input buffer
        LD A,(HL)       ;and first character
        CP ØDh          ;check for input of ENTER only(1ØØØ=76h)
        JR Z,Error      ;and goto error routine if found
        SUB "Ø          ;good character-make it binary
        PUSH HL         ;save pointer
        LD DE,Ø         ;set-up for first run through loop
        LD B,Ø          ;set-up for later use in BC
Mult1Ø  LD C,A          ;save current digit
        EX DE,HL        ;retrieve 'last value' of converted
                        ;binary number-note: it is Ø at first
                        ;and place it into HL
        ADD HL,HL       ;double it
        LD D,H          ;store HL*2 in DE
        LD E,L
        ADD HL,HL       ;double again
        ADD HL,HL       ;and one last time
        ADD HL,DE       ;adding HL*2 means HL=HL*1Ø
        ADD HL,BC       ;add the current digit
        EX DE,HL        ;temporarily store 'last value'
        POP HL          ;retrieve pointer
        INC HL          ;adjust it
        LD A,(HL)       ;get next digit
        CP ØDh          ;check for terminating ENTER (1ØØØ=76h)
        RET Z           ;and exit if found
        SUB "Ø          ;good character-make it binary
        PUSH HL         ;save pointer
        JR Mult1Ø       ;loop back to multiply by 1Ø
;
Error   LD HL,InpErr    ;load bad input msg and
        CALL Print      ;print it
        SCF             ;signal error occurred
        RET             ;return to main routine
;
;Program Messages
;
Renum   DEFB ØDh        ;1ØØØ=76h
        DEFM "Renumber "
        DEFB FFh
FromLn  DEFM "from line #:"
        DEFB FFh
NewLn   DEFB ØDh        ;1ØØØ=76h
        DEFM "Start with new line #:"
        DEFB FFh
Incr    DEFM "in steps of:"
        DEFB FFh
NotFnd  DEFB ØDh        ;1ØØØ=76h
        DEFM "Sorry, I cannot locate the line"
        DEFM "to renumber from!!"
        DEFB ØDh,FFh    ;1ØØØ=76h,FFh
InpErr  DEFB ØDh        ;1ØØØ=76h
        DEFM "****Invalid Input--Try Again****"
        DEFB ØDh,FFh    ;1ØØØ=76h,FFh
;
```

```
;Program Variables
;
OldLine DEFW Ø
NewLine DEFW Ø
Step    DEFW Ø
Pointr  DEFW Ø
Buffer  DEFB ØDh        ;1ØØØ=76h
```